



QUALITY OBJECTS (QUO)  
[HTTP://QUO.BBN.COM](http://quo.bbn.com)



# **QuO Toolkit**

## ***Users' Guide***



**QuO Release 3.1.0**

**September 2004**

---

## QuO Toolkit Release History

September 2004	QuO release 3.1.0
October 2002	QuO release 3.0.11
April 2002	QuO release 3.0.10
December 2001	QuO release 3.0.9
October 2001	QuO release 3.0.8
October 2001	QuO release 3.0.7
September 2001	QuO release 3.0.6
July 2001	QuO release 3.0.5
June 2001	QuO release 3.0.4
May 2001	QuO release 3.0
December 1999	QuO release 2.1
May 1999	QuO release 2.0
September 1998	QuO release 1.0

## Document History

September 2004	changes for release 3.1.0
October 2002	changes for release 3.0.11
April 2002	changes for release 3.0.10
7 December 2001	changed for release 3.0.9
30 October 2001	changed for release 3.0.8
28 September 2001	updates for release 3.0.7
4 September 2001	final updates for release 3.0.6
29 August 2001	Updates for release 3.0.6
30 July 2001	updates for release 3.0.5
26 June 2001	Updates from 3.0.3 to 3.0.4
06 June 2001	Reformatting to produce better HTML output
18 May 2001	QuO 3.0 release

### *On the cover:*

The Eastern Quoll is a carnivorous marsupial found in Tasmania. See:  
<http://www.parks.tas.gov.au/wildlife/mammals/equoll.html>  
Copyright © 2004 by BBN Technologies.

Produced by BBN Technologies.  
10 Moulton Street, Cambridge, MA 02138, (617) 873-8000.

# CONTENTS

---

Preface.....	v
Purpose of this Document .....	v
Background of the Project.....	v
Technical Questions .....	v
1 Introduction .....	7
1.1 New features of this release.....	8
1.2 How QuO Enhances a Distributed Application .....	8
1.3 Roles for Developing Adaptive Distributed Applications .....	10
1.4 Qoskets: Support for Reusing Systemic Behavior .....	11
1.5 Summarizing Features of QuO – Qoskets, Delegates .....	13
2 Installing QuO .....	14
2.1 Components of the QuO Toolkit.....	14
2.1.1 Components Explained .....	15
2.2 Installing QuO from Sources.....	15
2.3 Running Examples and Services .....	16
2.4 Types of ORB Supported .....	16
2.5 A Brief Note on Makefiles .....	16
3 Setting Up Services .....	17
3.1 StatusTEC (Status Typed Event Channel) .....	17
3.1.1 Description .....	17
3.1.2 Software .....	17
3.2 RSS (Resource Status Service) .....	18
3.2.1 Description .....	18

---

3.2.2	Software .....	18
3.3	DiffServ Service .....	19
3.3.1	Description .....	19
3.3.2	Software .....	19
4	QuO Example Applications .....	20
4.1	Suggested Tour of the examples .....	20
4.2	Simple .....	21
4.2.1	Goals.....	21
4.2.2	Description .....	22
4.3	Methodrtt.....	22
4.3.1	Goals.....	22
4.3.2	Description .....	22
4.4	Local-Translator .....	22
4.4.1	Goals.....	22
4.4.2	Description .....	22
4.5	Bette .....	23
4.5.1	Goals.....	23
4.5.2	Description .....	23
4.6	Bette-rmi.....	23
4.6.1	Goals.....	23
4.6.2	Description .....	24
4.7	Bette-local .....	24
4.7.1	Goals.....	24
4.7.2	Description .....	24
4.8	simple_ciao.....	24
4.8.1	Goals.....	24
4.8.2	Description .....	24
4.9	simple_mico .....	25
4.9.1	Goals.....	25

4.9.2	Description .....	25
4.10	diffserv_ciao.....	25
4.10.1	Goals.....	25
4.10.2	Description .....	25
5	Adding QuO to an Application .....	26
6	Attaching Delegates .....	31
6.1	Client Side.....	31
6.2	On The Server Side .....	33
7	Creating Delegates .....	35
7.1	A Note About the Application-Specific Adaptive Behavior.....	41
7.2	Next Steps .....	42
8	Creating Qoskets .....	43
8.1	Creating Qosket Objects.....	43
8.1.1	Defining the Qosket interfaces .....	44
8.1.2	Write the Qosket Implementation .....	44
8.2	Creating a Qosket Component .....	44
8.2.1	Defining a Qosket Component's Interface .....	45
8.2.2	Writing a Qosket Component Implementation .....	45
8.3	Defining a Contract .....	46
8.4	ASL .....	46
8.5	Property Files .....	47
8.6	Makefile .....	47
8.6.1	Makefiles for Qosket Objects.....	47
8.6.2	Makefiles for Qosket Components.....	47
8.7	Qosket Inheritance.....	47
8.7.1	Qosket Object Inheritance .....	47
8.7.2	Qosket Component Inheritance .....	48
9	Resource Status Service (RSS) .....	49
9.1	Motivation .....	49

---

9.2	RSS Architecture.....	51
9.2.1	Data Feed.....	53
9.2.2	Resource Contexts.....	53
9.2.3	Data Formula.....	54
9.2.4	Data Value.....	54
9.3	QuO Resource Ontology .....	55
9.4	Setting up the RSS .....	55
9.4.1	Setting up Resource Configuration Servers .....	56
9.4.2	Setting up StatusTEC .....	56
10	Integrating QuO Services .....	57
10.1	Creating CORBA Servers .....	57
10.2	Creating SysConds .....	57
10.3	RSS Integration Points .....	57

## **Preface**

### **Purpose of this Document**

This manual is a guide to implementing quality-of-service for distributed applications using the QuO Toolkit.

This manual is part of a set of evolving manuals that consists of a *Users' Guide* and a *Reference Guide*. At this time, this documentation is not as complete as the QuO software. More information will be provided in future releases of the documentation. In the meantime, additional information about how to build and run the software is in the README files in the directories of the release.

### **Background of the Project**

QuO is sponsored by DARPA and was initially funded partially by Quorum program. The work reported in this release is in part funded by the DARPA PCES and in part by the DRAPA AIRES program. These programs, seek to ensure end-to-end quality of service for distributed applications.

BBN Technologies' Quality Objects (QuO) research is extending the distributed object middleware paradigm to enable the development of adaptive distributed applications.

### **Technical Questions**

Please direct any technical questions to [quo-users@bbn.com](mailto:quo-users@bbn.com).

---

*This page is blank intentionally.*

# 1 Introduction

QuO is an application-development framework for developing distributed applications that control and adapt to dynamically changing network, CPU and data conditions, and Quality of Service (QoS) requirements. QuO employs distributed object and component middleware and provides various tools and utilities for providing end-to-end QoS in a distributed application. The ultimate benefits of such adaptive distributed systems are efficiency and predictability under dynamically changing conditions.

QuO extends traditional middleware in several directions to help programmers create and reuse adaptive code. The basic approach is to separate the adaptive code from the base functionality of the application. This can either be done at object level or at component level by designing adaptive components separately from functional or business components. Because of this separation, the adaptive code can be maintained independently of the base application and can be reused across several applications. Managing this separation is a difficult task for which the programmer needs extensive support throughout the application's life cycle. QuO strives to provide this support. Currently, developers tend to spread adaptive behavior throughout an application, thus adaptive code becomes *tangled* with the business logic of the program. QuO uses several techniques to tease out the adaptive behavior, such as Aspect Oriented Programming (AOP), code generation, reflectivity, open implementation, encapsulation, distributed services, and libraries. QuO supports realizing this adaptive code as objects and components so that QuO can be used with applications developed using CORBA 3.0 or prior versions.

Specifically, QuO supports the development of applications that can perform the following:

- *Specify the level of service they desire.* That is, in addition to specifying functional interfaces, an application can specify its systemic (non-functional) requirements, such as real-time response, memory utilization, access control, synchronization, managed bandwidth, or dependability.
- *Specify behavior alternatives and strategies for adapting to changing levels of service.* An application can choose different behaviors based upon its current operating mode, the level of service being provided by the system, *etc.* These adaptive behaviors can be used to recover from problems, to proceed in the face of degraded service, to implement different processing modes, and to enhance portability among different configurations.
- *Measure and control system resources, conditions, and mechanisms.* QuO provides feedback to the application about the state of the system and provides standard interfaces (called *system condition objects* or *sysconds*) to resources and mechanisms. The application can use the feedback information to trigger adaptation or reconfiguration. Likewise, the application can use the control interfaces to try to achieve the desired service.

---

These capabilities go beyond those provided by conventional distributed object middleware. The QuO framework consists of the following components:

- Description languages for describing QoS contracts in a distributed system, adaptive behavior alternatives and strategies, and probes into the system to measure and control QoS mechanisms and resources.
- Code generators that create and distribute executable (currently Java and C++) code throughout the application and system.
- A runtime kernel that organizes, schedules, and manages contract evaluation, feedback, and updates.
- Reusable libraries of system condition objects that interface to system resources, mechanisms, and managers.
- An encapsulation model, *Qoskets*, for encapsulating QoS adaptive behaviors and instantiating them as runtime components, *qosket* components, that can be assembled with the functional components, and reusable libraries of *qoskets* and *qosket* components. that provide data adaptations (scaling, compression, frame-rate, fragmentation, pacing, defragmentation, decompression), network adaptations (Diffserv) and CPU adaptations (CPU reservations).

### 1.1 New features of this release

In this QuO 3.1 release, dynamic and adaptive QuO support is provided to applications that use the CORBA Component Model of CORBA 3.0. This is achieved by encapsulating adaptive QoS behaviors as components, called *qosket components*. These *qosket* components extend the *qosket* object functionality offered in QuO 3.0 and can be developed separately from functional components, can be configured with the application components using CCM tools, and can adapt the behavior of the system at run-time.

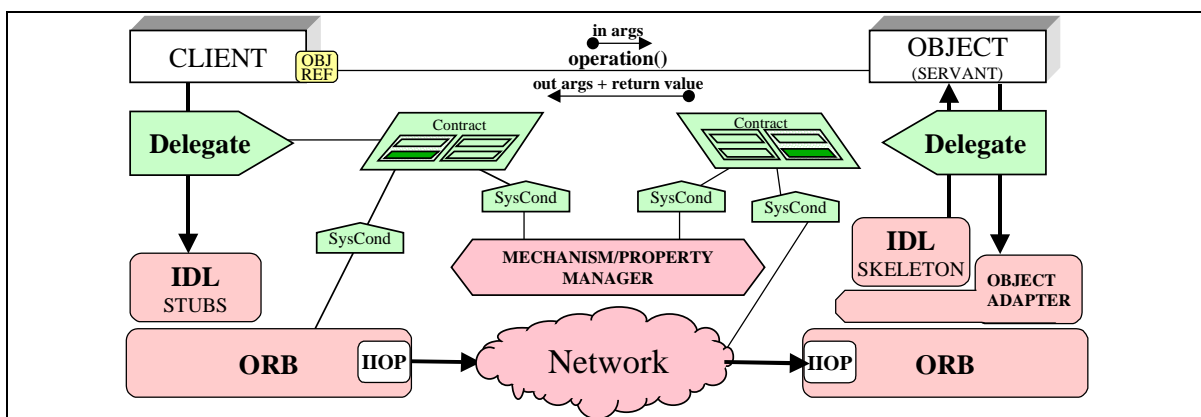
In addition, QuO 3.1 includes new *qoskets*, new examples, and bug fixes. Some of the examples, *qoskets*, and mechanism support from earlier versions, such as RSVP and OODTE, have not been actively maintained and have been omitted from this release.

### 1.2 How QuO Enhances a Distributed Application

In a traditional distributed object application using the CORBA model, a client makes a method call on a remote object through its functional interface. The arguments to the call are marshaled by an ORB on the client's host, delivered using a standard protocol (such as CORBA's IIOP) to an ORB on the remote object's host. The remote ORB unmarshals the data and delivers it to the remote object, which executes the method. The method's out parameters and return value, if any, are marshaled by the remote ORB, delivered to the client's local ORB via the standard protocol, unmarshaled by the client local ORB, and delivered to the client.

In a distributed component application using CORBA 3.0's CORBA Component Model (CCM), the same basic flow of control takes place. However, marshalling and unmarshalling of data, references to servants and ORBs, and POA management is encapsulated in a container. CCM allows object interfaces to be encapsulated into a component abstraction, which includes not only (possibly multiple) object interfaces, but also defines life cycle control. CCM provides a standard way of designing components, configuring the connections of these components and their default attributes at assembly time, packaging these components as distributable units, and deploying them over the network.

The CORBA Component Model allows object interfaces to be encapsulated into a component abstraction, which includes not only (possibly multiple) object interfaces, but also defines life cycle control. CCM provides a standard way of designing components, configuring the connections of these components and their default attributes at assembly time, packaging these components as distributable units, and deploying them over the network.



*QuO adds components to control, measure, and adapt to QoS aspects of an application*

At the object level, QuO supports the ability to insert the following into an application, as illustrated in the figure above:

**QuO contracts** provide control, adaptation, and reaction decisions in the application. They summarize an application's current operating mode, expected resource utilization, rules for transition among operating states, and means for notifying the application or system of changes in QoS or in system status. Contract specifications are written in a high-level specification language (not raw Java or C++ code) and preprocessed into compilable code-by-code generators. Contracts can operate as simple reflections of system state (data), reactive controllers (compensators), supervisory controllers, and more.

**System Condition objects (SysConds)** provide object interfaces to system resources, mechanisms, and managers. They provide standard, high-level, and reusable interfaces to measure, manipulate, and control lower-level real-time control and measurement capabilities. They export values that describe facets of system status, such as the current memory utilization or priority of the running thread, and provide interfaces to control system characteristics, such as modifying the processor clock rate or scheduling priorities.

---

**QoS-aware Delegates** are adaptive wrappers that modify the system's runtime behavior along the paths of method calls, returns, and event delivery (collectively exposed as *ports* in CCM). These help QuO support code adaptation triggered by contract region transitions or by instrumentation packages for measuring performance.

Collections of contracts, system condition objects, and delegates making up a particular adaptive behavior or QoS concern can be collected into a *Qosket* encapsulation and instantiated as qosket components, as described in Section 1.4 below, for assembly into component applications.

More information about QuO, including copies of published papers and QuO V3.1 software, can be found at: <http://quo.bbn.com>.

### 1.3 Roles for Developing Adaptive Distributed Applications

A key principle behind QuO, CCM, and related middleware activities is the separation of concerns and lifecycle support for the roles in the creation and fielding of a robust software system. The QuO middleware supports the separation of the *functional* and *QoS* concerns and software processes define the epochs of *requirements*, *design*, *development*, and *maintenance*. CCM further decomposes the *development* epoch into *implementation*, *packaging*, *assembly*, and *deployment*.

Even though one developer may play multiple roles, it is important to split up the development process functionally, in order to clarify the boundaries between the various components. The roles are:

The **Application Functionality Developer**, who develops the business logic of an application and realizes it as objects or components.

The **Qosketeer**, who develops reusable adaptive code (Qoskets).

The **Adaptation Integrator**, who merges or assembles the adaptation code (qosket components or qosket objects) into the business logic code.

The **Mechanism Developer**, who wraps services with CORBA and CCM servers and SysConds.

The **Application User**, who uses the adaptive application.

The **Installer**, who downloads and installs software, and verifies the install.

The **System Administrator**, who sets up services.

## 1.4 Qoskets: Support for Reusing Systemic Behavior

The QuO architecture has many mechanisms for inserting adaptive code throughout the system. This section discusses adaptive code, which resides above the ORB and how to reuse the adaptive code, using an architectural element called a *Qosket*.

Above the ORB adaptation consists of layers, each made up of a Delegate and a Contract with SysConds. The Delegate intercepts all method calls to a remote object and adapts its interactions to the remote object based on the current QoS Region given by the Contract. The Delegate is the hook for *in-band* control between objects that works on the *data-plane*, much like a layer in a network protocols stack or the wrapper pattern. The Contract with its System Conditions is *out-of band* in the *control-plane*, much like a manager, which monitors and controls the systems QoS, but does no actual (in-band) work and does not get in the way. The interaction between the data-plane and control-plane is restricted to the interaction between Delegate and Contract

This architecture does a nice job of separating the in-band concerns from the out-of-band concerns. But this architecture only partially addresses the separation of the business behavior from the systemic behavior. Business behavior tends to reside in the Delegate and the systemic behavior tends to reside in the Contract and SysConds. But the system behavior may need to insert some code in-band, such as monitoring code to measure response time. Also, the business behavior may need more knobs for controlling the system than just querying the Contract for its QoS Region.

Qoskets provide a unit of reuse for the QuO Version 3.1 Framework. A Qosket is defined *independently* of any business interfaces for objects and components. Thus, a Qosket definition can be reused for many different applications. The Qosket definition contains the traditional CDL for the Contract and SysConds, but may also contain adaptive code that will be woven into the Delegate. A Qosket is specialized to a particular business interface to make a Delegate for the business interface of a remote object. The Delegate has both the business behavior of the remote object and the adaptive behavior of the Qosket. A qosket can be instantiated as a particular component or set of components to be assembled into a functional component assembly.

A major reuse challenge for the QuO is that the adaptive code in the Delegate depends on both business adaptation and systemic adaptation. The systemic adaptation needs to be disentangled in order to reuse the systemic adaptation in many business domains. In order to do this, we use several state of the art approaches to pull out the system code and bundle it into a reusable specification for the adaptive code. First, Aspect Oriented Programming (AOP) is used to define code that will be woven into the Delegate. Second, code generators are used to create the Contract object and stubs that interface the Delegate with the contract object. Third, open-implementation techniques are used to define *hooks* for adding in business adaptation. Fourth, components are used to encapsulate qosket elements and assembly tools are used to insert them into component applications. Lastly, helper libraries are defined to make writing the business adaptation easier. These techniques strive to be heterogeneous across distributed middleware systems (CORBA and RMI) and languages (C++ and Java).

---

In summary, a Qosket is a bundle of specifications that can be woven together with a business adaptation specification to generate a Delegate and Contract layer. The Qosket exposes an interface to the client programmer for monitoring and controlling the systemic adaptation. The Qosket also exposes an interface to the business adaptation code to define helper functions to make writing the business adaptation easier. The Qosket also includes native language implementations for these interfaces. Lastly, the Qosket can include the systemic adaptation that will go into the Delegate, which are described using an Aspect Oriented Programming language call ASL. The Qosket specification can be pre-generated into a library of native language objects, which are integrated into the Delegate when it is generated, or instantiated as qosket components that are assembled using off-the-shelf assembly tools.

Qoskets can be used in two ways. To use a Qosket with an object application, the adaptation integrator must define some business adaptation and generate a Delegate. The business adaptation includes specifying the business adaptation code that will go into the Delegate using the ASL language. Also, users can write a wrapper to simplify the interface to the Delegate. The business specification and the Qosket specification are generated into the Delegate, which will be used by the client developer.

To use a Delegate, the client developer instantiates a specific QuO Delegate class. The Delegate instance acts like a portal through which the client and object interactions have a specific kind of QoS adaptive behavior. Since the Delegate is created from a specific Qosket, it has a specific kind of adaptive behavior. Another Delegate to the same remote object can use a different Qosket and will have a different kind of adaptive behavior. Both Delegates perform the same business function, but adapt in different ways. The client developer can instantiate both Delegates and call them in different parts of the code. For example, suppose one Qosket optimizes throughput, e.g. using a pre-fetch pattern, and another Qosket optimizes latency. Then the throughput Delegate (generated from the throughput Qosket) could be called inside a loop and the latency Delegate could be used outside the loop. This three phase process of creating a Qosket, creating a Delegate and using the Delegate is further described in the section 8.

To use a Qosket with a component application, the adaptation integrator instantiates the Qosket as a set of qosket components and assembles them into the functional assembly. A Qosket Component not only provides the same functionality as an object-based qosket instantiation but also provides the benefits of a CORBA Component. That is, it provides a standard format for implementing components (by abstracting away repetitive and error prone programming of POA-specific code); life cycle and run-time support from the container in which it will run; and the ability to be configured, assembled and deployed with the other components using standard assembly and deployment tools. Thus, a qosket component adds adaptive and dynamic behavior to an application by being assembled with the application's business components.

The addition of component support working with qoskets is one of the main additions of QuO Verson 3.1. The prototype of *qosket components* in QuO Version 3.1 makes two advances in the QuO toolkit. First, it provides a version of QuO that works with CORBA 3.0 and CCM. Second, it provides a basis for composable QoS adaptive behaviors. Qosket components are realizations of Qoskets that can be composed, or assembled.

## **1.5 Summarizing Features of QuO – Qoskets, Delegates**

A goal of QuO Version 3.1 is to provide dynamic and adaptive QoS support to distributed applications developed using the CORBA Component Model just as the goal of QuO 3.0 was to be able to reuse systemic behavior independent of business behavior. This means not only there is a clean separation of concerns between business behavior and systemic behavior; in order to support the reuse of systemic behavior, but also the ability to assemble and deploy this QoS behavior as a set of qosket components exists.

---

## 2 Installing QuO

### 2.1 Components of the QuO Toolkit

The QuO release consists of the components indicated by checkmarks in the table below:

Component	Linux i386 src Tarball	MacOS src Tarball	Windows2000 src ZIP
QuoCore	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
QuoDoc	<input checked="" type="checkbox"/>	-	-

All these components can be downloaded as open-source at <http://quo.bbn.com>.

### 2.1.1 Components Explained

QuoCore	The QuO base system including <ul style="list-style-type: none"> <li>- QuO runtime Java and C++ Kernel</li> <li>- QuO codegenerator (quogen) with support for CCM Components</li> <li>- Status Typed Event Channel Service (STEC)</li> <li>- Resource Status Service (RSS)</li> <li>- Demonstration qoskets and examples</li> </ul>
QuoDoc	Written and generated documentation for QuO 3.1

## 2.2 Installing QuO from Sources

QuO sources are released as a set of tarballs. After downloading the tarballs, unpack them by

```
> tar xfvz quo*.src.tgz
```

The quoCore component must be built before any other components are built. For **building the quoCore component**, set the QUO\_ROOT environment variable to the top-level directory where you unpacked the component:

```
/home/user> tar xfvz quoCore.src.tgz
```

and setup your environment (using bash) by

```
/home/user> export QUO_ROOT=/home/user/quo
```

For **running/building most other QuO components**, you have to set component specific variables in addition to QUO\_ROOT. See Chapter 8 of the QUO Toolkit Reference Guide for more details. You can use the following script to get an initial environment:

```
/home/user> source $QUO_ROOT/scripts/environment/quoenv.sh
```

To start the compilation process, do a

```
/home/user> cd $QUO_ROOT
/home/user> make quoCore quoDoc
```

---

## 2.3 Running Examples and Services

After installing QuO, we recommend running examples as described in the Section 4.1 Some of the examples demonstrates the instantiation of the QuO Services explained in the Section 3

## 2.4 Types of ORB Supported

There are five types of ORBs supported – three for C++ and two for Java

C++ ORB: TAO, CIAO and MICO are the three C++ ORBs.

Java ORB: Jacorb and JDK are the Java ORBs. For this release, JDK ORB is used.

## 2.5 A Brief Note on Makefiles

The QuO Makefiles follow existing coding standards as close as possible.

For C++, there are two kinds of makefiles. One type is structured according to TAO makefile paradigm and other is modeled after GNU makefiles. The former needs to use “-DQUO\_USE\_TAO” flag and the later needs “-DQUO\_USE\_MICO” where TAO and MICO are the specific ORBs for which these makefiles are designed for.

For Java, easy to understand makefiles using standard utilities (javac, GNU utilities) were written.

For QuO with CIAO, ACE's new project management system, MPC (The Makefile, Project and Workspace Creator) is used. For more details, please refer to

[http://www.cs.wustl.edu/~schmidt/ACE\\_wrappers/MPC/README](http://www.cs.wustl.edu/~schmidt/ACE_wrappers/MPC/README). To generate the GNU Makefiles or Visual Studio workspace files, change the current working directory to `$QUO_ROOT/examples/simple_ciao`, then execute `'$ACE_ROOT/bin/mwc.pl simple.mwc'`. Use the '-type' argument to mwc.pl to specify the desired output (default is GNU Makefiles). See `$ACE_ROOT/MPC/README` and `$ACE_ROOT/MPC/USAGE` for more information. Also notice how mpc is used with codegen to generate code that will be used for qosket components.

For using QuO in an application that uses CORBA Objects, select the preferred makefile paradigm. However, modeling the makefiles for QuO specific parts (*i.e.*, QDL code generation) based on the makefiles contained in the release is recommended. For a detailed description of the makefile structure for `$QUO_ROOT/examples/simple`, and `$QUO_ROOT/examples/simple_ciao`, see the *Quo Toolkit Reference Guide*.

## 3 Setting Up Services

The QuO release contains a set of prebuilt services. A *service* is a facility that provides QuO with the use of something. Examples of services are managers for bandwidth reservation, access control, system resource information, and intrusion detection. In order to be used by QuO, a service has to interface with QuO by means of qoskets that might contain system conditions, callbacks, and contracts. In many cases, components providing a service already exist as COTS software. The QuO release does *not* contain those third party components, but provides wrappers (named qoskets) to utilize them in QuO enabled applications.

The following sections describe services integrated into the QuO release.

### 3.1 StatusTEC (Status Typed Event Channel)

#### 3.1.1 Description

QuO TEC is a Java implementation of the CORBA Typed Event Service. QuO TEC has flexible policies, which allow it to be configured either as a traditional "event" channel or as a "status" channel. QuO uses the Status TEC configuration to implement a Resource Status Distribution System, which publishes status information about host and network resources to applications.

The QuO StatusTEC is intended to scale to an Enterprise network service, *i.e.*, spread over several sites connected by a Wide Area network. A mesh of Status TEC channels act as one logic channel/service. Producers and consumers of status information can start and stop, independent of each other and their location in the network.

#### 3.1.2 Software

The StatusTEC service is included in the quoCore component of QuO 3.1. The BWSelectWrapper in the bette-rmi example demonstrates its use.

---

## **3.2 RSS (Resource Status Service)**

### **3.2.1 Description**

The QuO Resource Status Service (RSS) is a unified way to access information about the status of system resources. Instances of an RSS are local to the application, both on the client-side and on the object-side. The RSS gives the application an integrated view of all of its resources and keeps the view as up-to-date and as consistent as possible. Applications can query the RSS for current status information and can also subscribe to changes in status.

QuO Version 3.1 supports the following types of Data Feeds. The Property Data Feed can remotely load property files for the base configuration of the system. The Status TEC Feed subscribes to a QuO Status Typed Event Channel, which uses push technology to publish the status of hosts.

### **3.2.2 Software**

RSS is included in the quoCore component of QuO 3.1. The RSSWrapper in the bette example demonstrates its use.

### **3.3 DiffServ Service**

#### **3.3.1 Description**

DiffServ is a protocol for providing network level adaptation by setting codepoints on IP header for TCP/UDP/SCTP. These packets are then prioritized by being put in different queues by the routers configured with codepoints that is set on the IP packets. QuO provide DiffServ service by encapsulating this as a DiffServ qosket component. Users can select any priority between 0 –32767 using a system condition exposed on the component as an attribute, this then subsequently gets mapped to network priority using the Priority Mapping Manager. It is possible to set priority at the ORB level (all invocations from this ORB will have the codepoints set), Object level (all the invocations from the object will have the codepoints set) and thread level (All invocations from a thread will have diffserv codepoints set) on the client side and at POA and ORB level at the server side. In our current implementation, we provide the ability to set diffserv codepoint at the ORB level using RTORB in the RTComponentServer.

#### **3.3.2 Software**

RTComponentServer required comes with ACE/TAO/CIAO release. `diffserv_ciao` example demonstrates the use of DiffServ in prioritizing network traffic.

---

## 4 QuO Example Applications

The quoCore component contains a set of examples that demonstrate how to use QuO in applications. The examples are meant to augment the overall documentation by source code that can be compiled and executed. The focus is not so much on demonstrating the example application itself, but rather on highlighting and demonstrating QuO specific features for both CORBA Objects and CORBA Components as specified by CORBA Component Model (CCM). By understanding the examples, user will learn how to:

- Encode adaptive behavior in QDL (ASL, CDL)
- Implement and reuse system conditions and callbacks
- Handle QuO's complexity with makefiles
- Compile and run QuO enabled applications
- Interface QuO with a 3rd party product
- Add QuO to an already existing application
- Design adaptive behavior to be reusable
- Encapsulate adaptive behavior as qosket CCM components
- Generate makefiles using MPC for qoskets and generating code for codegen

### 4.1 Suggested Tour of the examples

There are two ways to proceed the tour of examples: examples that demonstrate QuO adaptation using CORBA Objects and the examples that demonstrate QuO adaptation using CCM Components. Use the table below to find the best example.

Using CORBA Objects	Purpose	example
	get a basic understanding of QuO	simple

	interested in the C++ CORBA part of QuO	Methodrtrt
	interested in the C++ part of QuO with no middleware	local-translator
	to see the most complex use of QuO featuring many standard qoskets	Bette
Using CCM Components	to learn to write qosket	simple_ciao
	to evaluate the differences in writing qosket using CIAO vs MICO	simple_mico
	a more complex example demonstrating the real-time adaptation (setting diffserv codepoints for prioritizing network traffic) encapsulation in a qosket,	diffserv_ciao

To run any of the following examples, you must first setup your Shell Environment (see *QuO Toolkit Reference Guide*).

The following sections describe examples that are in \$QUO\_ROOT/examples.

## 4.2 Simple

### 4.2.1 Goals

This example teaches how to write a qosket containing a customized callback and basic contract, and hook it up to an already existing application. Simple is special in that it is the only example that runs across all supported middleware platforms (Java CORBA, Java RMI, C++ CORBA). It also has the widest spread on operating systems (Linux, Solaris, Windows 2000). The focus is on understanding the basic QuO components.

---

### **4.2.2 Description**

"Simple" is a simple example of the use of QuO in an application. The server for this application is a Counter that maintains one piece of data, an integer variable. The 'count(x)' method adds an integer (passed as the parameter x) to the value at the server; the 'countDown(x)' method subtracts an integer from the server's value. The contract at the application client controls whether the client's calls actually increment the server, decrement the server, or does nothing.

For details on how to build and run the example see \$QUO\_ROOT/examples/simple/README.

## **4.3 Methodrtrt**

### **4.3.1 Goals**

By understanding the Methodrtrt example, a user will learn how to use QuO in a C++ only application. This example is less complex than "Simple" since it only supports the C++ CORBA middleware platform. The focus is on understanding the basic QuO components.

### **4.3.2 Description**

The Methodrtrt Qosket adds the capability of measuring round trip time delays for the client's method calls on the server. It is based on the same application logic as Simple.

For details on how to build and run the example see \$QUO\_ROOT/examples/methodrtrt/README.

## **4.4 Local-Translator**

### **4.4.1 Goals**

Local-Translator is an example of how to use QuO in a C++ application with no middleware. A description of how to develop such an application appears in the local-translator/README file.

### **4.4.2 Description**

Local-Translator implements a Spanish-English dictionary with caching. When the application is run, one of four commands can be entered at the command line as follows:

- ts *word* – translates an English word to Spanish
- te *word* – translates a Spanish word to English
- add *English Spanish* – adds a word to the dictionary

done - exits the application

With QuO in the application, the delegate intercepts calls to the dictionary server. For a translation command, the delegate first looks in the cache to see if it can do the translation locally. If so, it returns the translation to the client. If not, it calls the server to do the translation, adds the new information to the cache, and returns the translation to the client. For the add command, the delegate adds the new entry to the cache and passes the information on to the server.

The communication between client and server is through standard C++ method calls.

For details on how to build and run the example see `$QUO_ROOT/examples/local-translator/README`.

## **4.5 Bette**

### **4.5.1 Goals**

bette shows how to use QuO's RSS service in an application. The example also shows how to combine multiple qoskets, and define local qoskets that are not shared. The focus of the bette example is on showing useful functionality through QuO services.

### **4.5.2 Description**

This example is intended to demonstrate a simple adaptation to resource loading. The client requests images from the server and displays them on the screen. (The images provided in the QuO distribution happen to be photographs of Bette Davis.) The Bette example defines three local client-side contracts. It also uses the client contracts defined by the RSS and Count qoskets, and the server contract defined by the Instrumentation qosket. The three local client contracts are Diagnose, which tries to determine whether or not a bottleneck exists and the source of the bottleneck if so (server, network, or unknown); UserAdapt, which allows for manual adaptation; and State, which demonstrates using a contract as a state machine.

For details on how to build and run the example see `$QUO_ROOT/examples/bette/README`.

## **4.6 Bette-rmi**

### **4.6.1 Goals**

Bette-rmi shows how to use the count, RSS, and bw\_select qoskets in Java RMI application.

---

## **4.6.2 Description**

This example is intended to demonstrate a simple adaptation to resource loading. The client requests images from the server and displays them on the screen. (The images provided in the QuO distribution happen to be photographs of Bette Davis.) The Bette-Rmi example uses the client contracts defined by the RSS, BWSelect and Count qoskets, and the server contract defined by the Instrumentation qosket.

For details on how to build and run the example see \$QUO\_ROOT/examples/bette-rmi/README.

## **4.7 Bette-local**

### **4.7.1 Goals**

Bette-local is an example of how to use QuO in an application where the communication between client and server does not go over a middleware (CORBA, RMI), but rather is done via local method calls.

### **4.7.2 Description**

This example is intended to the use of QuO technology in a local calling context, with no distributed object middleware. It's a simplified version of the bette examples in which the "client" and "server" are two local objects that communicate with ordinary Java calls. The QuO delegate simply counts the calls.

For details on how to build and run the example see \$QUO\_ROOT/examples/bette-local/README.

## **4.8 simple\_ciao**

### **4.8.1 Goals**

simple\_ciao has the same functionality as the Simple example (Section 4.2). However, it is the simplest example demonstrating a CCM qosket component – that is how can a CCM qosket component be implemented, how a adaptive behavior is achieved just by assembling and deploying these CCM qosket components along with functional CCM component (Client and Counter) using standard assembly and deployment tools. This example is only in C++ and requires CIAO.

### **4.8.2 Description**

The functionality is exactly the same as that of Simple example. However, all the SysConds are exposed as attributes and thus can be configured at run time.

## **4.9 simple\_mico**

### **4.9.1 Goals**

This example is exactly the same as simple\_ciao except for the fact that it demonstrated the differences one need to be aware of for designing and implementing qoskets using MICO instead of CIAO. Additionally, one can use assembly and deployment tool of MICO to assemble and deploy the CCM component assembly.

### **4.9.2 Description**

The functionality is exactly the same as simple\_ciao.

## **4.10 diffserv\_ciao**

### **4.10.1 Goals**

diffserv\_ciao example demonstrates how a real-time sophisticated network service can be encapsulated in a qosket component. And how the assembly of this qosket can provide network adaptations.

### **4.10.2 Description**

The functionality of this example is similar to the simple or simple\_ciao or simple\_mico with a minor difference – it provides the ability to set 3 diffserv codepoints at the ORB level on the TCP/IP packets. These three diffserv codepoints are expedited forwarding, assured forwarding and best effort services. The codepoints are set by qosket upon receiving a callback. This requires one to use RTComponentServer of CIAO.

---

## 5 Adding QuO to an Application

This section describes the general issues involving the introduction of QoS aware adaptation to a CORBA application by using the QuO toolkit. Details on how to perform the different steps and how to create various QuO components involved in the process are described in this and the next two chapters. For information on how to add QuO to a local application (no middleware), see the file `$QUO_ROOT/examples/local-translator/Local_QuO.txt`.

CORBA is used here as a representative example of the more general DOC (Distributed Object Computing) paradigm. The concepts underlying QuO are applicable to DOC in general, and currently QuO supports CORBA (both Objects and CCM Components) and Java RMI, the two widely used DOC implementations and the CCM.

Development of a CORBA application starts with defining interfaces in IDL. IDL compilers are then used to produce the stub, skeleton, and helper classes that will be used in defining the objects or components that implement the interfaces and the objects or components that use them.

In a typical CORBA scenario, objects interact through stubs and skeletons. A client that invokes an operation on a remote object is essentially interacting with a local stub that handles the remote operation in a transparent manner. Similarly, an object that implements the operation interacts with a local skeleton object that transparently handles requests from all its clients. If application uses CCM components, same flow of control takes place except that container is added as another layer of abstraction. The container provides life cycle control to the application components and performs routine exercises like initializing the ORB, interacting with POA and communicating with stubs and skeletons.

The components such as client and server objects/CCM components are referred as business components, and the functionality implemented by these objects as the business aspect of the application. Using the QuO toolkit you can define components that capture QoS awareness, QoS control and QoS oriented adaptive behavior; and more importantly, incorporate these QoS aspects with the business aspect of an application.

The following common usage cases exist for developing applications using CORBA Objects:

- Traditional CORBA approach

The traditional CORBA approach is for the object-developer to create an object and just publish its IDL. The client-developer takes the IDL and generates client-side proxies and uses basic CORBA to access the remote-object.

- Library approach

The library approach is for the object-developer to create a library that pre-generates the CORBA proxies and in addition wraps them to simplify the interface or actually install code on the client-side. The client-developer can just use the familiar library pattern and does not have to worry about generating CORBA.

The following common usage cases exist for developing applications using CORBA (CCM) Components

- Shared Library approach

The Usage case for applications using CCM Components solely depends on shared library approach. A shared library is created for each component and it contains component's implementation (executor) stubs, skeleton, and servants. The path to these libraries is specified in the XML-based descriptors that are used in the assembly and deployment of these components.

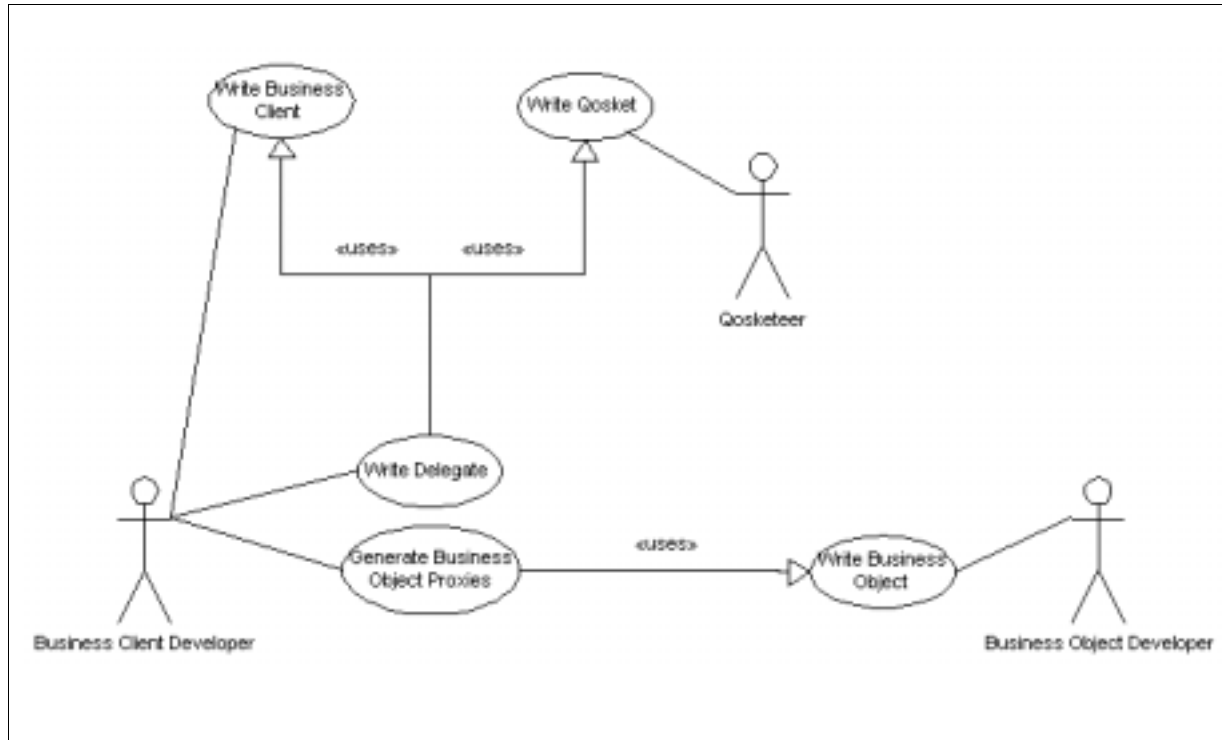
In order to introduce QoS awareness and adaptive behavior, this set up is modified slightly: in case of CORBA Objects, instead of interacting with a stub or a skeleton the objects now interact with what we call a QuO Delegate. A QuO Delegate merges a reusable module of QoS aspects known as a Qosket with adaptive behavior that is specific to the user of the QuO Delegate (*i.e.*, the client or the server object). In case of CCM components, QuO Delegate is encapsulated as a part of separate component, called as Qosket Component.

A Qosket (see also: Creating Qoskets) definition may contain:

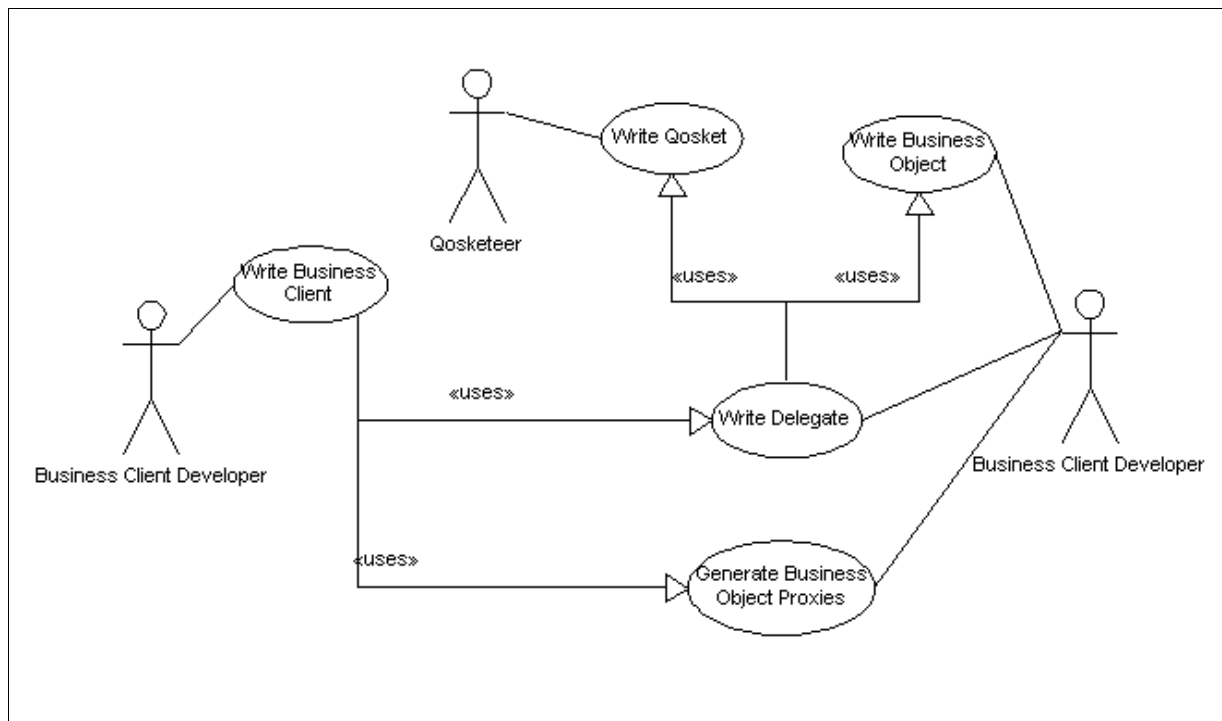
- Specification of a QuO contract in CDL (Contract Description Language),
- Specification of adaptive behavior in ASL (Aspect-oriented Structure-description Language)
- IDL interfaces for system conditions and callbacks needed in the contract, and
- Programming language classes providing the implementation of system conditions and callbacks

Adaptive behavior associated with the user of the QuO Delegate is expressed in ASL. If the user of the QuO Delegate is a client object then this ASL description is tied with the remote calls it makes. If the user is a server object, then this ASL is tied with the operations that it makes available to remote clients.

Analogous to the different CORBA use cases described earlier in this chapter for applications based on CORBA Objects, QuO can be added to an already existing application in two ways, as shown in the figure below:



*CORBA-Object-style Use Case for QuO*



*Library-style Use Case for QuO*

In the first style, whoever needs to use the services of an object develops the Delegate and in the second style, the developers of the object exposes the object through a Delegate. Although the current release does not contain any example that is based on the library style, it is possible to use the QuO toolkit to expose an object that displays QoS aware adaptation through a Delegate.

QuO can be added to the applications just by assembling and deploying Qosket components along with business components for applications using CCM components. The assembly and deployment is based on XML-based descriptors as specified by Deployment and Configuration specification of OMG.

No matter what the use case is, introduction of QoS aware adaptation to CORBA Objects based application involves one of the following scenarios:

- Reuse both Qosket and Delegate

This is arguably the simplest way to introduce QoS aware adaptation to your application: the user needs to attach the Delegate to the application. The chapter Attaching Delegates describes the procedure for doing this. This is more relevant for the library style use of QuO, where the object developer wants to expose the business aspects implemented by the object in a QoS aware adaptation enabled manner. As mentioned earlier, this release of QuO has basic support for reuse of both Qoskets and Delegates, but it has not been incorporated in the set of services that are part of this release. However, the \$QUO\_ROOT/examples/bette example comes pretty close. This example builds several Delegates that a slide client can connect to.

- Reuse a Qosket, but Write the Delegate

This is the preferred approach to make use of the QuO services to meet one's QoS aware and adaptation needs. The implementers of the QuO services provide the Qosket, and whoever needs to use the service creates the QuO Delegate customized to use the service in their specific application. This way one can add application specific behavior to the service. The chapter Creating Delegates describes the procedure for creating the delegates. Once the user has the delegates, the user can attach them to the application as described in Attaching Delegates.

- Write both Qosket and Delegate

This is the most advanced way of using QuO. Creating Qoskets is the preferred approach of QuO service developers to expose their QoS aspects. This can take various forms: one can start from scratch, or take an existing Qosket and change its components, or can collect different components from different Qoskets and modify them etc. Once the Qosket is available and the business object developer wants to expose one's object in a QoS aware and adaptation enabled manner (following the library style), one can create the delegate and expose it. Otherwise, the business client developer creates one's own Delegate making use of the Qosket to access the services of the remote object in a QoS aware and adaptation enabled manner. The chapter Creating Qoskets describes the mechanism for creating a Qosket.

---

Introduction of QoS aware adaptation to CCM components based application involves:

- Reuse Qosket Component

A Qosket component is comprised of two aspects – encapsulation of the adaptive behavior and interaction with business components. The former is the unit of reuse and can be reused by any number of applications. The components that comprise this unit of reuse are further described in Section 8. The latter needs to be modified so that a qosket component mirrors the interfaces of the business components that it is interacting with. That is it provides the same interface to a server component that a client component was providing and a same interface to a client component that a server component was providing. This enables a qosket component to get assembled in between the business components and provide adaptation to the application transparently.

## 6 Attaching Delegates

This section describes how to attach a Delegate to an application, assuming that such a Delegate exists. Depending on who the user of the Delegate is (*i.e.*, a client accessing a remote object through the delegate or a server exposing itself through the delegate) and the implementation language (*i.e.*, Java or C++) there are multiple cases, as shown below. This chapter shows how the code looked originally and then shows what was added or changed in order to attach the Delegate. The '+' prefix indicates new code, and the 'c' prefix means changed code.

### 6.1 Client Side

Attaching a Delegate on the client-side is straightforward. Instead of accessing the remote object directly, the client is changed to access the remote object through the looking glass of the Delegate. The interconnection of QoS machinery gets hooked into the client and the QoS aspects affect the way the client interacts with the remote object.

Instead of connecting to a stub (simplest way to do so is by resolving an IOR), the client connects to a Delegate. This is done by instantiating a Delegate, and invoking an initialization method on it. All subsequent remote calls can then be made on this Delegate.

The following code snippet demonstrates the change in client code (taken from \$QUO\_ROOT/examples/simple), where the Delegate is an instance of the native language class SwitchedCounterWrapper that has an initialization routine `_quo_initialize(..., ...)`.

C++ Business: client.cpp

```
-----
....

CORBA::ORB_var orb = ...;
...
// Remote server of IDL type Counter
CORBA::Object_ptr optr = orb->string_to_object (...);
Counter_var remote_counter = Counter::_narrow(optr);
...
//Now use the remote object
remote_counter->count(1);
...
```

C++ QuO Enabled Business: client.cpp

```
-----
....

CORBA::ORB_var orb = ...;
...
// Remote server of IDL type Counter
CORBA::Object_ptr optr = orb->string_to_object (...);
```

---

```

    Counter_var remote_counter = Counter::_narrow(optr);
+   // Create the QuO Delegate
+   SwitchedCounterWrapper *serv_wrap = new SwitchedCounterWrapper;
+   serv_wrap->quo_initialize (CORBA::ORB::_duplicate(orb.in()),
remote_counter);
    ...

    // Now use the Delegate to invoke remote operations
c   serv_wrap->count(1);
    ...

```

Java Business: Client.Java

```

-----
    ....

    ORB orb = ...;
    ...
    // Remote server of IDL type Counter
    CORBA::Object optr = orb.string_to_object (...);
    Counter remote_counter = Counter::_narrow(optr);
    ...

    //Now use the remote server
    remote_counter->count(1);
    ...

```

Java QuO Enabled Business: Client.Java

```

-----
    ....

    ORB orb = ...;
    ...
    // Remote server of IDL type Counter
    CORBA::Object optr = orb.string_to_object (...);
    Counter remote_counter = Counter::_narrow(optr);
+   //Create the QuO Delegate
+   SwitchedCounterWrapper serv_wrap = new SwitchedCounterWrapper();
+   serv_wrap._quo_initialize (orb, remote_counter);
    ...

    //Now use the Delegate to invoke remote operation
c   serv_wrap->count(1);
    ...

```

## 6.2 On The Server Side

Attaching a Delegate on the server-side raises additional issues. In some sense, it amounts to putting an object behind the looking glass of the Delegate. The interconnected QoS machinery wraps around the server object and presents itself as the provider of the same service. This means that the server's original object reference is different from its QuO enabled object reference. The Delegate fronts as the server object that it wraps and intercepts the calls meant for the wrapped object. This is done passing a non-remote reference to the CORBA object that is being wrapped to the initialization method of the Delegate.

The following code fragments show how delegates are attached on the server side. The example is written in Java, the object being wrapped is of IDL type `SlideServer` implemented by the Java class `SlideShowServer`. The Delegate is of IDL type `SlideShowInstrumented`, implemented by the Java class `SlideShowInstrumentedServerWrapper` with an initialization method called `connect(..., ..., ...)`.

Java Business: Server.java

-----

```

...
main (...) {
  ORB orb = ...;
  POA poa = ...;
  ...
  // instantiate the server
  SlideShowServer  real_server= new SlideShowServer(...);

  // create CORBA object that you want to register with the ORB,
  // write IOR etc
  SlideShow reference = real_server._this();
  ...
  orb.run();
}

```

Java QuO Enabled Business: Server.java

-----

```

...
main (...) {
  ORB orb = ...;
  POA poa = ...;
  ...
  // instantiate the server to be wrapped
  SlideShowServer  real_server= new SlideShowServer(...);

+  // instantiate and connect the Delegate
+  SlideShowInstrumentedServerWrapper wrapper = null;
+  wrapper = new SlideShowInstrumentedServerWrapper(.....);
+  wrapper.connect(orb,server,.....);

  // create CORBA object that you want to register with the ORB,
  // write IOR etc
c  SlideShowInstrumented reference = wrapper._this();

```

---

```
    ...  
    orb.run();  
}
```

Note that it is possible to have the Delegate and the wrapped object to have different interfaces and it is also possible to expose both the wrapped object and the wrapper as legitimate targets of remote calls.

## 7 Creating Delegates

A Delegate is a fundamental QuO construct that merges business and QoS aspects. Conceptually, it could be thought of as a runtime entity that gets in the middle of object invocation path and has interconnection to the QuO machinery responsible for QoS awareness and control. This chapter describes how to write a Delegate. For details on how to write qoskets see Creating Qoskets.

A prepackaged Qosket, generally speaking, defines a set of contract regions and behavior associated with these regions. The regions and region-specific behavior is reusable, and is not associated with any particular business logic. It is not necessary that all Qoskets have region-specific behavior. The QuO 3.0 release is shipped with a library of Qoskets, the SwitchQosket defines several contract regions that are determined by a user controlled value but does not specify any particular behavior associated with the regions.

Creating a delegate consist of the following steps:

1. Specify application-specific adaptive behavior in ASL:

Application-specific adaptive behavior (based on the contract regions defined in the Qosket) is specified in ASL. The ASL description in the delegate might reuse ASL specifications from the qosket (for instance, \$QUO\_ROOT/examples/methodrtt/adapter/qdl/MethodRTT.asl uses an ASL template defined in qosket/methodrtt/qdl/MethodRTT\_Template.asl).

The following ASL file from \$QUO\_ROOT/examples/simple specifies adaptive behavior that involves changing a remote call to a different one depending on what region the contract is currently in.

File: \$QUO\_ROOT/examples/simple/adapter/qdl/CounterDelegate.asl

```
-----
//Count is the business interface
behavior Count ()
{

    // The target of the remote calls
    remote_object Counter remote;

    //How to adapt invocation of method in the business interface
    long Counter::count(in long arg) {
        return_value long count;

        //Adaptation is inserted when a call takes place
        inplaceof METHODCALL {

            // These regions are defined in the contract
            //included in the Switch Qosket
            region DoNothing {
                count = 0;
            }
        }
    }
}
```

---

```

    region Decrement {
        count = remote.countDown(arg);
    }

    region Increment {
        count = remote.count(arg);
    }
}

// Force post-method contract evaluation.
before POSTMETHODCONTRACTEVAL {
}

};

```

For a more detailed understanding of this ASL description, see the `$QUO_ROOT/examples/simple` and *QuO Toolkit Reference Guide*.

## 2. Specify the combined Business-Qosket interface in IDL:

The next step is to create an IDL interface that combines the business and the QoS aspects, which serves as the basis for implementing the Delegate. The user can easily derive the combined business qosket interface by using inheritance in IDL.

```

File: $QUO_ROOT/examples/simple/adapter/idl/SwitchedCounter.idl
-----
#include "../business/idl/Counter.idl"
#include "../qosket/idl/SwitchQosket.idl"

interface SwitchedCounter : Counter , SwitchQosket
{
};

```

It is also possible to add additional methods on to the business qosket interface. Then the user has the responsibility to provide the implementation of these methods in a wrapper class (described below).

## 3. Run quogen to generate an adaptive proxy and adapter class:

This release comes with a makefile that performs the various QuO code generation steps and native language compilation steps to produce executables. Refer to the makefile documentation in the *QuO Toolkit Reference Guide* for further information on how to run quogen to produce the adaptive proxy and the adapter class.

The adaptive proxy can be thought of as a combination of business logic and application-specific adaptation. The adapter class can be thought of as the adaptive proxy that knows how to interact with the QoS awareness and control mechanisms. The adapter class provides the implementation of the merged interface described above.

The QoS awareness and control mechanisms represent an interconnected framework of various QuO components. In earlier releases, this interconnection was set up in part by CSL (now deprecated) and native language code. Some of the components in the interconnected framework are placeholders for real objects that can only be found at a later (runtime) stage. However, the mechanism to correctly interconnect them when they become available is encoded in the framework. The QuO runtime is one such example. The actual remote object (if the QuO machinery is being inserted in the client side, then it is the target of the client's remote operations; if the machinery is being inserted in the server side, then it is the object that wants to be available through the QoS infrastructure) that the QoS machinery wraps around is another. Other examples may include other application-specific objects that are used to support contract regions and adaptive behavior such as other remote objects to which application's remote calls can be dispatched or objects that can be used to receive callbacks from QuO Runtime. This is the reason that we need to take the next step.

4. Implement a wrapper class that wraps the adapter:

The users of the Delegate will access the Delegate through an instance of this class. Upon instantiation, an initialization routine is expected to be called. The initialization routine is responsible for filling in the placeholders mentioned above. Some of the references used in this processes can be passed in input arguments of the initialization routine and it will have to obtain the rest by other means. The wrapper class is the place where one has to define the implementation of the additional methods one has defined in the merged Qosket-Business interface.

The following code fragment explains the structure of a general Wrapper class in Java and C++. In both cases, note that:

- The name of the QuOWrapper class is SwitchedCounterWrapper.
- The name of the (generated) QuOAdapter class is SwitchedCounterAdapter.
- The QuOWrapper uses a prepackaged Qosket named SwitchQosket, which has all the QuO System Condition objects and QuO Callback objects it needs.
- The actual remote object the QuOWrapper wraps around is of type Counter.
- The QuOWrapper exposes `_quo_initialize(...)` as the initialization routine, its two parameters being a reference to the orb and the actual remote Counter object.
- The finalization routine typically has five well-defined tasks that are described in the five sections of the routine.
- The QuoRuntime (QuoKernel) is started as a separate process (actually as a CORBA object) and is accessed by reading in its IOR.

---

## In Java:

File:

```
$QUO_ROOT/examples/simple/adaptor/java/com/bbn/quoexamples/simple/Switch  
edCounterWrapper.java
```

```
-----  
-----  
public class SwitchedCounterWrapper extends SwitchedCounterAdapter  
{  
public void _quo_initialize (org.omg.CORBA.ORB orb, Counter server)  
{  
//SECTION 1: QuO Runtime  
//Declare a local variable to hold the QuoKernel  
QuoKernel kernel = null;  
//Obtain a reference to an externally started QuoKernel  
java.io.FileReader inFile;  
java.io.BufferedReader reader;  
String ior;  
try  
{  
inFile = new java.io.FileReader("QuoKernel.ior");  
reader = new java.io.BufferedReader(inFile);  
ior = reader.readLine();  
kernel = QuoKernelHelper.narrow(orb.string_to_object(ior));  
reader.close();  
} catch (Exception e) {}  
  
//SECTION 1a: Connector (Java CORBA only)  
// Connector is part of the QuO runtime for CORBA Java  
//implementation of QuO. It is a placeholder  
// for the current orb, making sure that it has the right object  
com.bbn.quo.corba.impl.Connector.orb = orb;  
//SECTION 2: System Conditions  
//Initialize and interconnect the sysconds wrt the  
//QuO Runtime i.e. QuoKernel  
initSysconds(kernel);  
  
//SECTION 3: Callbacks  
//Similarly, initialize and interconnect the callbacks  
initCallbacks();  
  
//SECTION 4: QuO Contract  
//Initialize the QuO Contract.  
linkContract(kernel);  
  
//SECTION 5: The actual remote object  
//Interconnect the actual remote object wrt the QuO Machinery  
linkRemoteObject(server);  
}  
}
```

**In C++**

The declaration (*i.e.*, the .h file) is shown first and then the implementation (*i.e.*, the .h file).

```
File: $QUO_ROOT/examples/simple/adapter/cpp/SwitchedCounterWrapper.h
```

```
-----  
//This is the declaration of the SwitchedCounterWrapper class
```

```
#include "SwitchedCounterAdapter.h"
```

```
class SwitchedCounterWrapper : public virtual  
com::bbn::quoexamples::simple::SwitchedCounterAdapter  
{  
public:  
  
    /// Constructor  
    SwitchedCounterWrapper();  
  
    /// Destructor  
    virtual ~SwitchedCounterWrapper();  
  
    /// QUO initialization function  
    virtual int _quo_initialize (const CORBA::ORB_ptr &orb,  
    const CORBA::Object_ptr &server);  
};
```

```
File: $QUO_ROOT/examples/simple/adapter/cpp/SwitchedCounterWrapper.cpp
```

```
-----  
//This is the implementation of the SwitchedCounterWrapper class
```

```
#include "SwitchedCounterWrapper.h"
```

```
//Constructor  
SwitchedCounterWrapper::SwitchedCounterWrapper() {}
```

```
//Destructor  
SwitchedCounterWrapper::~SwitchedCounterWrapper(){}
```

```
//The initialization routine  
int SwitchedCounterWrapper::_quo_initialize (  
const CORBA::ORB_ptr &orb,  
const CORBA::Object_ptr &server)
```

---

```

{
// Section 1: QuO Runtime
// Using ACE facility to read the IOR filename of the externally
//started QuoKernel
ACE_TString filename(ACE_TEXT("file://QuoKernel.ior"));
CORBA::Object_ptr obj = orb->string_to_object (filename.c_str());
quo::QuoKernel_ptr kernel = quo::QuoKernel::_narrow (obj);

// Section 2: QuO System Conditions
//Initialize and interconnect the System Condition objects wrt
//the QuO Runtime
initSysconds (quo::QuoKernel::_duplicate(kernel));

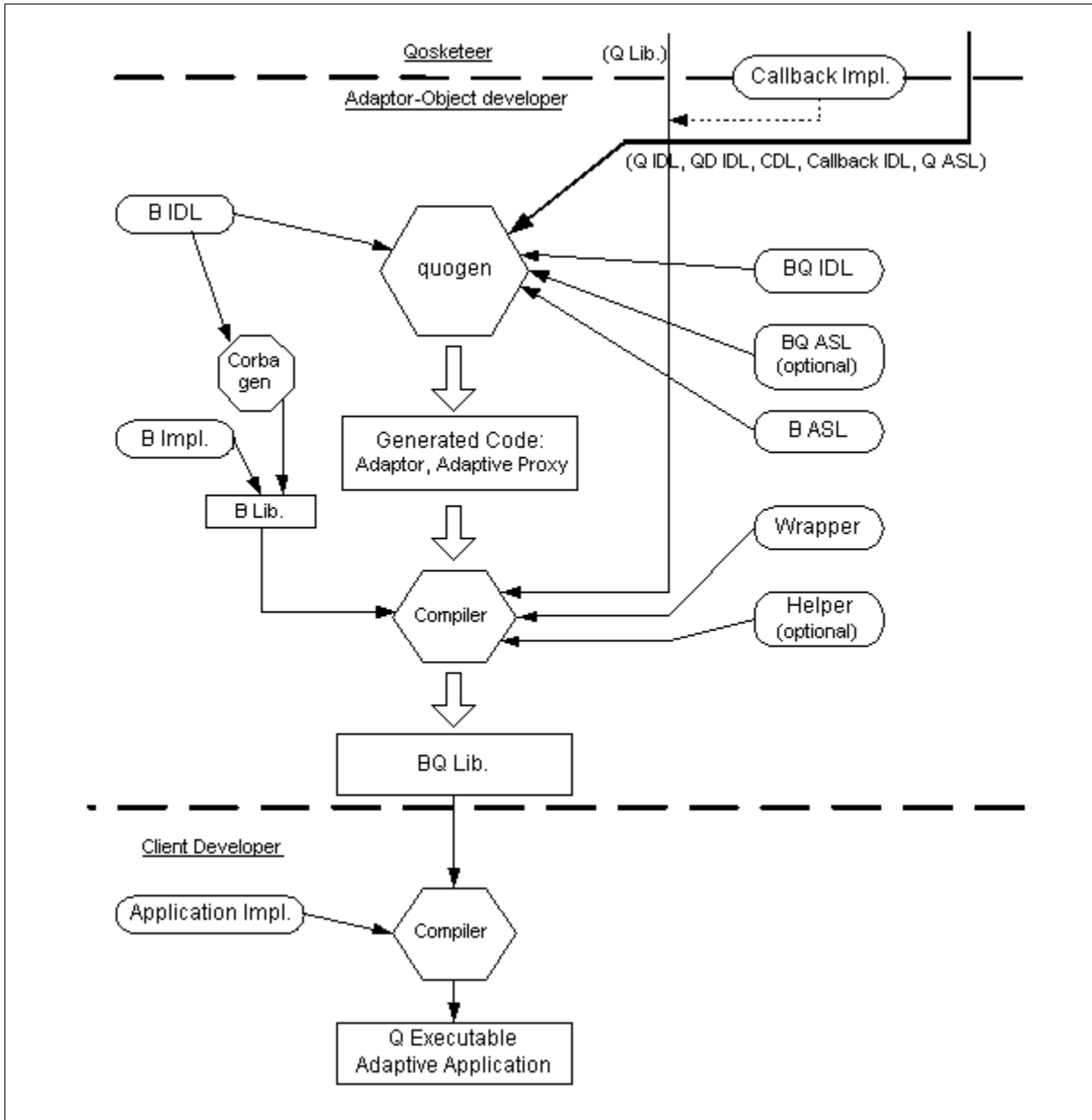
// Section 3: QuO Callbacks
// Initialize and interconnect the Callbacks wrt the QuO Runtime
initCallbacks (CORBA::ORB::_duplicate(orb));

// Section 4: QuO Contract
// Initialize the QuO Contract
linkContract (quo::QuoKernel::_duplicate(kernel));

// Section 5: The actual remote object
//Interconnect the actual remote object wrt the QuO machinery
linkRemoteObject (Counter::_duplicate(Counter::_narrow(server)));
}

```

5. Now that one has a QuOWrapper, one can follow the instructions for "using a prepackaged QuOWrapper" to use the wrapper created in the application code. The following figure depicts the various steps involved in developing a QuO application. The region within the dotted line relates to the process described in this section.



*Developing a QuO Application*

### 7.1 A Note About the Application-Specific Adaptive Behavior

This ASL specification ties the QoS aspects with the business aspects by putting the regions and region-specific behavior described in the Qosket in the context of the application. Some Qoskets need information from the application to function properly, this information (for instance, the size of an argument of a remote call) can only be obtained at run time. If this is needed the Qosket provides an interface, which should be used by the application-specific adaptive behavior specification.

---

The code to collect such information and invoking the Qosket's interface can in theory be part of the ASL that defines application-specific adaptive behavior. However, this chunk of ASL code is independent of the contract regions although they are specific to the interface associated with the application-specific adaptive behavior, and therefore, can be factored out into a reusable piece of ASL whose job is to collect and provide the information needed by the Qosket. `$QUO_ROOT/examples/bette` is an example that demonstrates this. The `SlideShow.asl` is a reusable ASL description that does not depend on any contract region, however it makes use of external variables and interfaces to pass information between the business logic and the Qosket.

## 7.2 Next Steps

You now know how to integrate existing QuO code into your application. The next steps from here involve writing your own QuO code, and this is the subject of the next section.

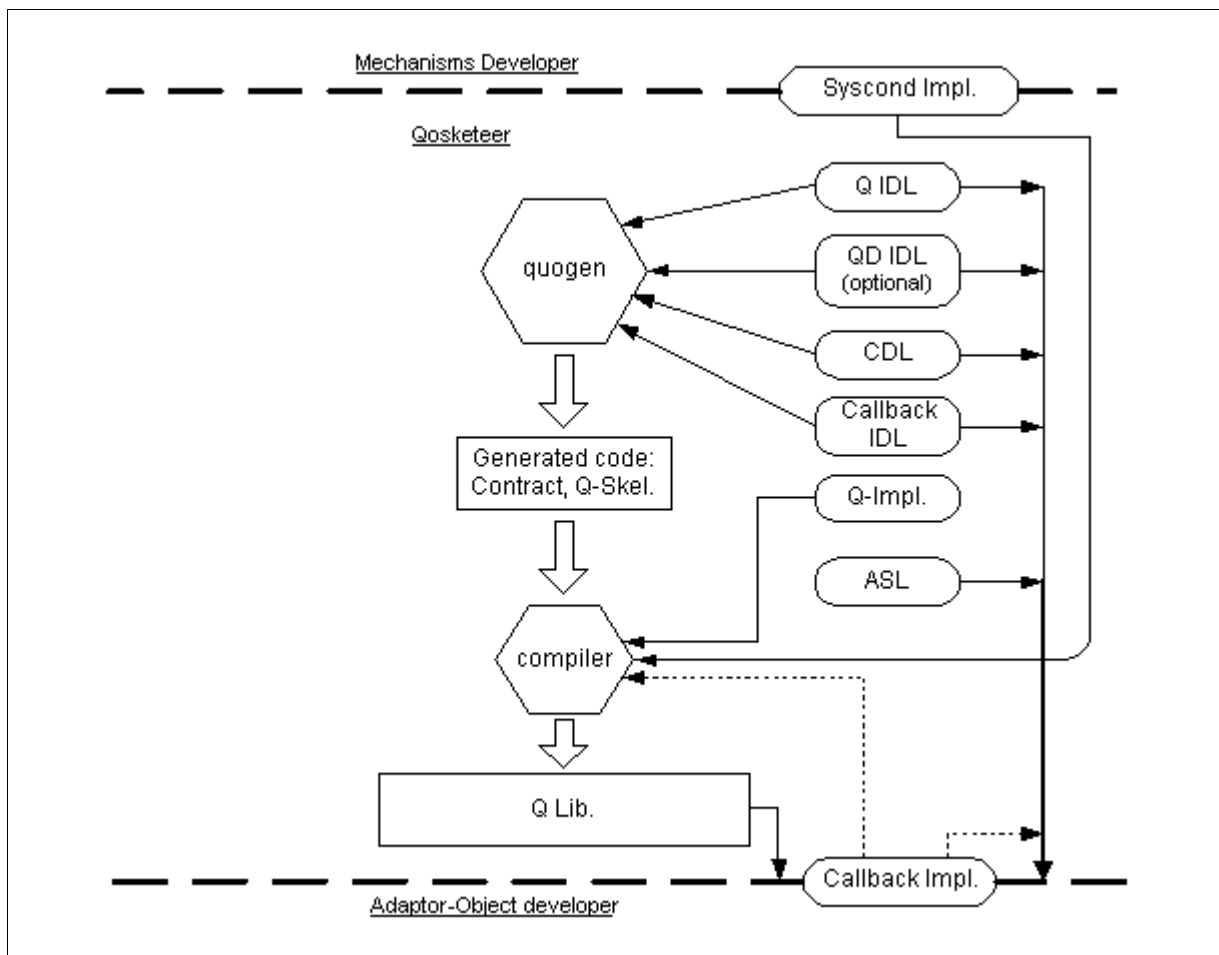
We would be very interested in hearing about other issues such as these as you encounter them; please send email to [quo-users@bbn.com](mailto:quo-users@bbn.com).

## 8 Creating Qoskets

There are two types of Qoskets: Qosket Objects and Qosket Components. The process of creating a Qosket Component encompasses the steps needed to create a Qosket Object and the steps needed to construct CORBA Component Model compliant modules termed as components.

### 8.1 Creating Qosket Objects

The basic job of writing a qosket can be described very simply: define the interfaces, write the implementations, define a contract, and optionally provide some generic adaptation with ASL or ASL templates. One can package sets of quogen command-line options into property files, to simplify the life of the users of your qosket, but this is more a matter of convenience than necessity. Finally a Makefile must be configured to automate the build process.



Qosket Development

---

These steps are described below, along with some preliminary notes on qosket inheritance (which is still at a primitive stage in QuO 3.0).

### **8.1.1 Defining the Qosket interfaces**

Qoskets must provide at least one interface, describing the methods that can be invoked from adapter or wrapper code. They may also provide a second interface, describing the methods that can be invoked from adaptive proxies (in ASL).

The specific content of these two interfaces is entirely up to you, but they have to be written in IDL. The QuO code generator quogen will translate the IDL description to native language description of the same interface. Note that one can't use the standard IDL->Java or IDL->C++ code generators in this regard, since the resulting classes would wind up as CORBA, which is not what they should be. Note also that, while the IDL must be syntactically valid, it doesn't have to be semantically legal. Non-CORBA types can be used in qosket IDL.

### **8.1.2 Write the Qosket Implementation**

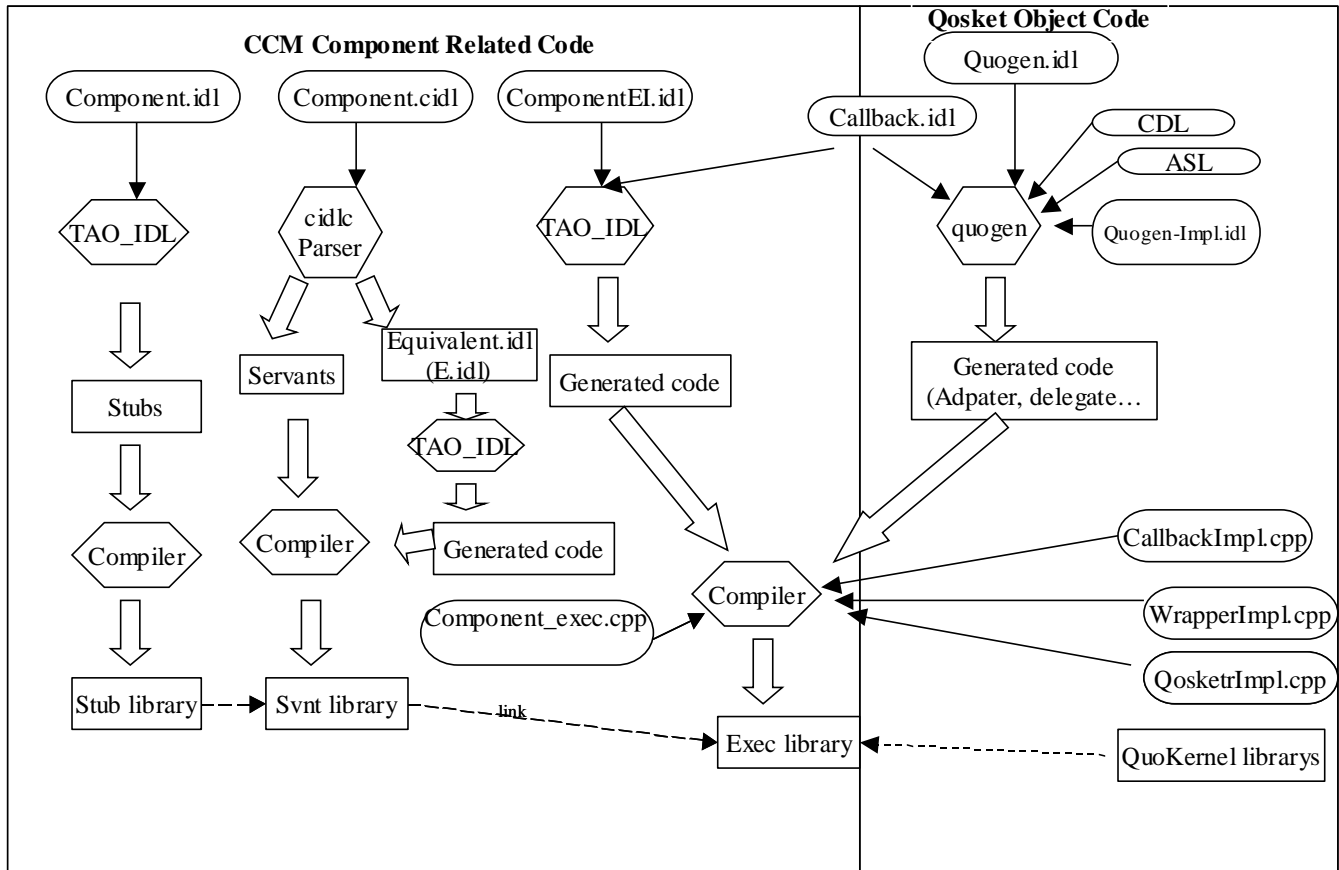
Given the abstract definitions expressed in the two interfaces, one must supply at least one native-language (C++ or Java) implementation. For Java, one may want to supply two versions, one for CORBA and another for RMI.

In either case, the implementation class must extend (Java) or derive from (C++) the qosket's generated skeleton class (MyQosketSkel, if the qosket is called MyQosket). In C++, an implementation class may of course derive from other classes as well. This can be convenient if there is a need to create a qosket that's a subclass of one or more other qoskets. In Java, this option is not available, but quogen provides a partial workaround for single-inheritance (see Qosket Inheritance).

## **8.2 Creating a Qosket Component**

The job of creating a Qosket Component includes (1) writing a CORBA Component Model compliant interface, (CIDL – defines composition and, EI.idl – defines the inheritance hierarchy and locality of the interfaces in case of CIAO) and (2) creating a CORBA 3.0 interface that quogen will use to generate code, providing the corresponding implementations, defining the contract, an optional ASL or ASL template. One can use the property files as in the construction of Qosket Objects. The TAO\_IDL compiler is needed to generate stubs and skeletons while the CIDL compiler is used to generate servant and executor classes. Quogen is used to generate adapter and delegates. Finally use MPC (provided with ACE) preferably to generate Makefiles. The process is outlined as (TBD,

### 8.2.1 Defining a Qosket Component's Interface



Developing a Qosket Component

One has to write two sets of interfaces, one needed for CORBA Component Model related interfaces and one needed to generate adaptive code. The former include a CORBA 3.0 compliant interfaces (Component.idl), and corresponding CIDL (defines the composition of interfaces, types of container associated) and optional EI.idl (defines locality of interfaces and their inheritance). The latter is exactly similar to those described in Section 8.1.1.

### 8.2.2 Writing a Qosket Component Implementation

For CCM Component related code, one needs to write a Component\_exec, QosketImpl, WrapperImpl and CallbackImpl (if callback is used) that contain implementations for all the interfaces defined.

---

Implementation of the `WrapperImpl` is modified for Qosket Component. It now inherits both from Qosket Adapter (as was in Qosket Object) and QosketComponent executor that is generated (and marked by CCM). The CCM components are packaged into executor, stub, and servant libraries, as defined by the CORBA standard. The CCM development environment will provide tools and Makefile scripts that build dynamically linked shared library files (\*.so files on Unix systems). Each component must provide two XML descriptor files containing relevant information

For each component, the CORBA component model expects one to provide a CORBA Software Descriptor (CSD) file, and a Servant Software Descriptor file (SSD) if one is using the CIAO implementation. These files provide entry point information for the shared library files and are used by the CIAO containers when connecting components together according to a Component Assembly Descriptor (CAD) file. To integrate a Qosket component into an existing CAD file, one needs to provide a `<componentfile>` tag that states the name of the CSD file. In the body of the CAD file, one can then determine in which process the component will be instantiated. Finally, within the `<connections>` region of the CAD file, create a `<connectevent>` tag that properly inserts the Qosket component between an existing interaction between two CCM components. For the CIAO implementation, one needs to also update the `CIAO_Installation_Data.ini` file to contain Universally Unique Identifiers (UUID) for the servant and executors of the components being assembled.

The current CIAO implementation of CCM uses a shared library base approach with session containers. Current extensions planned by the ACE/TAO/CIAO developers include the ability to use three types of containers (service, entity, and standalone) as well as the deployment of the executables in static libraries. We must caution developers that the error messages returned by CCM containers and assemblers often do not contain enough information to resolve problems that may occur at run-time. Be sure that the dynamically linked libraries have no undefined references, for example. Also make sure that within the CSD and SSD descriptor files, there are no typographical errors when entering the UUIDs.

### 8.3 Defining a Contract

Defining a qosket's Contract is described in *QuO Toolkit Reference Guide*. It is same for both Qosket Objects and Qosket Components. . It is the same for both Qosket Objects and Qosket Components.

### 8.4 ASL

Some qoskets can reasonably define generic adaptive behavior that would be of utility to the users of the qosket. These might be either ordinary ASL files or ASL templates. Many qoskets have no associated adaptive behavior, so this step is optional. Writing ASL is described in *QuO Reference Guide*. It is same for both Qosket Objects and Qosket Components. It is the same for both Qosket Objects and Qosket Components.

## 8.5 Property Files

For modularity reasons, it can be convenient to provide property files for a qosket, which define quogen options. One property file would generally have values for 'adapterqosket', 'delegateqosket', and 'contract', and would probably also include a 'qosketimpl' value. A user of the qosket would include this property file in the generation of an adapter/delegate. If you have two different implementation classes, one for CORBA and one for RMI, you would probably separate out the two 'qosketimpl' values into files of their own. If you expect your qosket to be subclassed, you may also wish to provide a property file defining the 'qosketbase' (see Property Files). It is same for both Qosket Objects and Qosket Components. It is the same for both Qosket Objects and Qosket Components.

## 8.6 Makefile

### 8.6.1 Makefiles for Qosket Objects

Aside from the usual compilation and packaging (jar or so file) steps, a qosket Makefile will always include a quogen step (see *QuO Toolkit Reference Guide*). If you're generating IDL from Java, it will also include a quoidlgen step (see *QuO Toolkit Reference Guide*). Here we mention only the issue of Makefile targets and dependencies.

The files generated for a qosket are the classes for the Contract (e.g., MyContract.java or MyContract.h) and the qosket skeleton (MyQosketSkel.java or MyQosketSkel.h). The simplest way to configure a make entry is to use one these generated files as the target, with dependencies on the IDL files, the CDL file and any property files you use in the quogen command. The operation is then simply the quogen command. See the Makefiles section for more details.

### 8.6.2 Makefiles for Qosket Components

Makefiles for Qosket Components is based on MPC and described in Section 2.5 and *QuO Toolkit Reference Guide*.

## 8.7 Qosket Inheritance

### 8.7.1 Qosket Object Inheritance

Qoskets can inherit from other qoskets. In general the process is simple. First, the qosket interface and adapter interface should be defined to extend the corresponding interface of the qoskets from which you wish to inherit. Second, the qosket implementation class should derive from the implementation classes of the qoskets from which you wish to inherit.

---

The complication comes with the second, in Java: since the qosket implementation class is already required to extend the qosket skeleton class, it can't extend anything else. For single inheritance, quogen provides a workaround: set the 'qosketbase' quogen option to the implementation class of the qosket from which you wish to inherit. This will define the generated qosket *\*skeleton\** class to extend the inherited implementation class. Since the local implementation class extends the skeleton class, it will now indirectly extend the inherited implementation class as well. For multiple inheritances in Java you'll have to write delegation methods manually.

### ***8.7.2 Qosket Component Inheritance***

Unlike multiple inheritance of interfaces, components follow a single inheritance – that is a component can inherit from only one component. However, so far we haven't really used component inheritance and so are not aware of any limitations, issues that one may encounter in doing so.

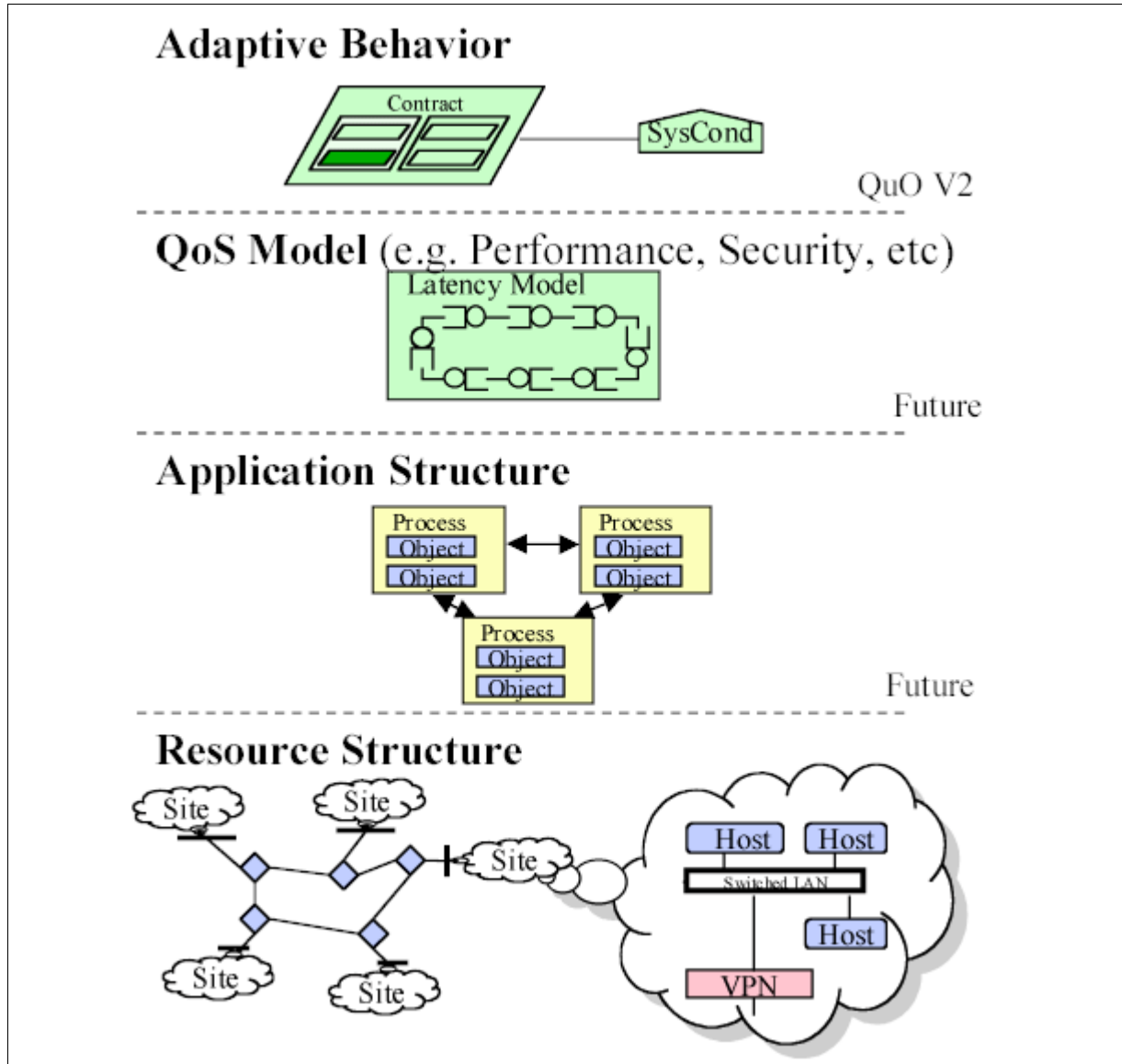
## 9 Resource Status Service (RSS)

The QuO Resource Status Service (RSS) is unified way to access information about the status of system resources. The RSS gives the application an integrated view of all of its resources and keeps the view as up-to-date and as consistent as possible. Applications can query the RSS for current status information and can also subscribe to changes in status. Clients can access the RSS either through direct Java calls (if the RSS is running in the same jvm) or through CORBA (if not).

QuO SysConds are the access points to the RSS, and their values can be part of predicates in a QuO Contract. The RSS is part of the QuO Java Kernel, and can be integrated into the client and object Java processes or run as a stand-alone process (*i.e.*, remote QuO Kernel) for use with C++ applications. QuO RSS, a new mechanism introduced in QuO Version 3.0, is ported to C++ and significantly simplified in this QuO version 3.1.

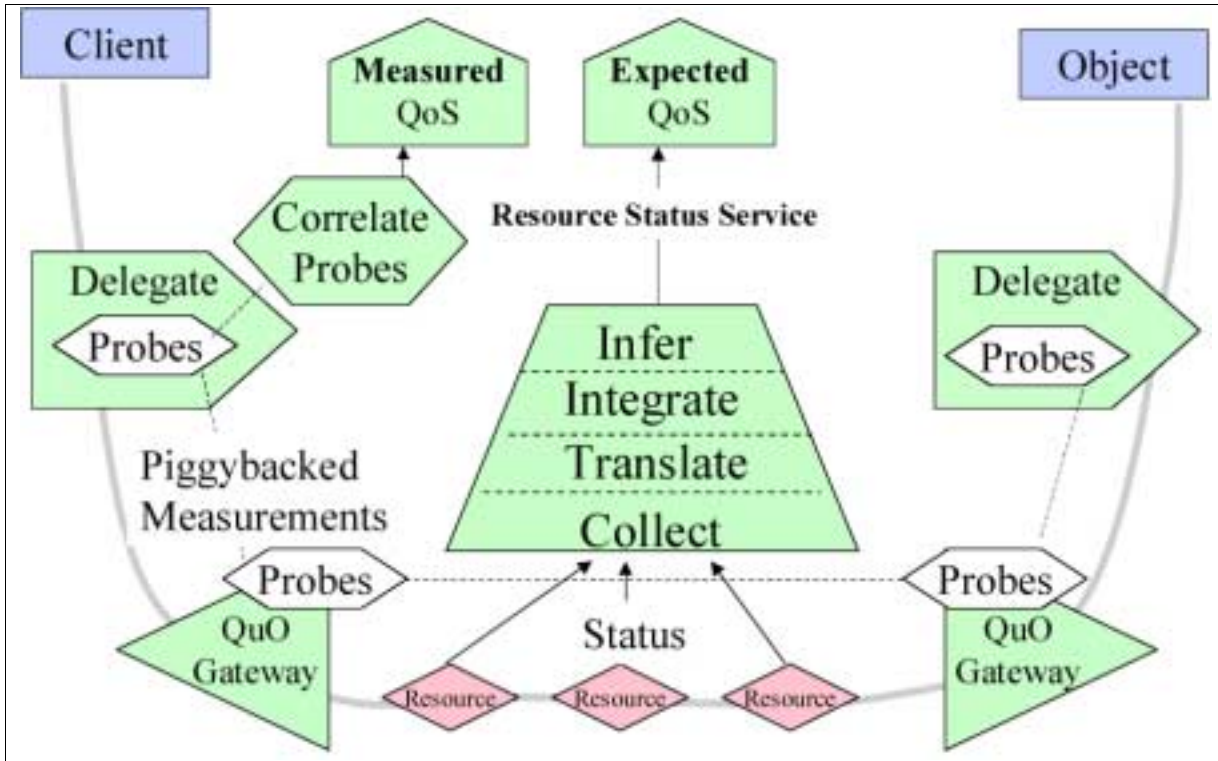
### 9.1 Motivation

Adaptive code needs to reason about the running system at several levels (see figure below). The highest-level adaptation would like to reason in terms of a QoS model of the system. This model depends on the application structure and the underlying resources. The more detailed the model and its inputs, the more precise adaptation can be. But short cuts and heuristics are possible and widely used in practice. The heuristics, which tend to be very specific to a given situation, can be based on information about the raw resources or local measurements of deliver QoS. We would like to make this information easily available.



*Adaptive Behavior Must Reason About the Running Application at Several Layers*

Adaptation tries to resolve the differences between what is observed and what is expected. At the heart of QoS adaptive code is a representation of the underlying system resources. QuO has two mechanisms for observing and inferring the status of resources (see figure below). First, in-band instrumentation is used to measure the timing and size of remote method calls. As the call travels from the client to the object, different probes add information to a trace-record that is piggybacked on the call. Since the trace record follows an individual call, it has a correlated view of the timings at different parts of the system. Second, out-band measurements collect information about the status of each resource on the path between the client and the server. Combining resource information along with the applications requirements for resources allows the expected performance to be inferred.



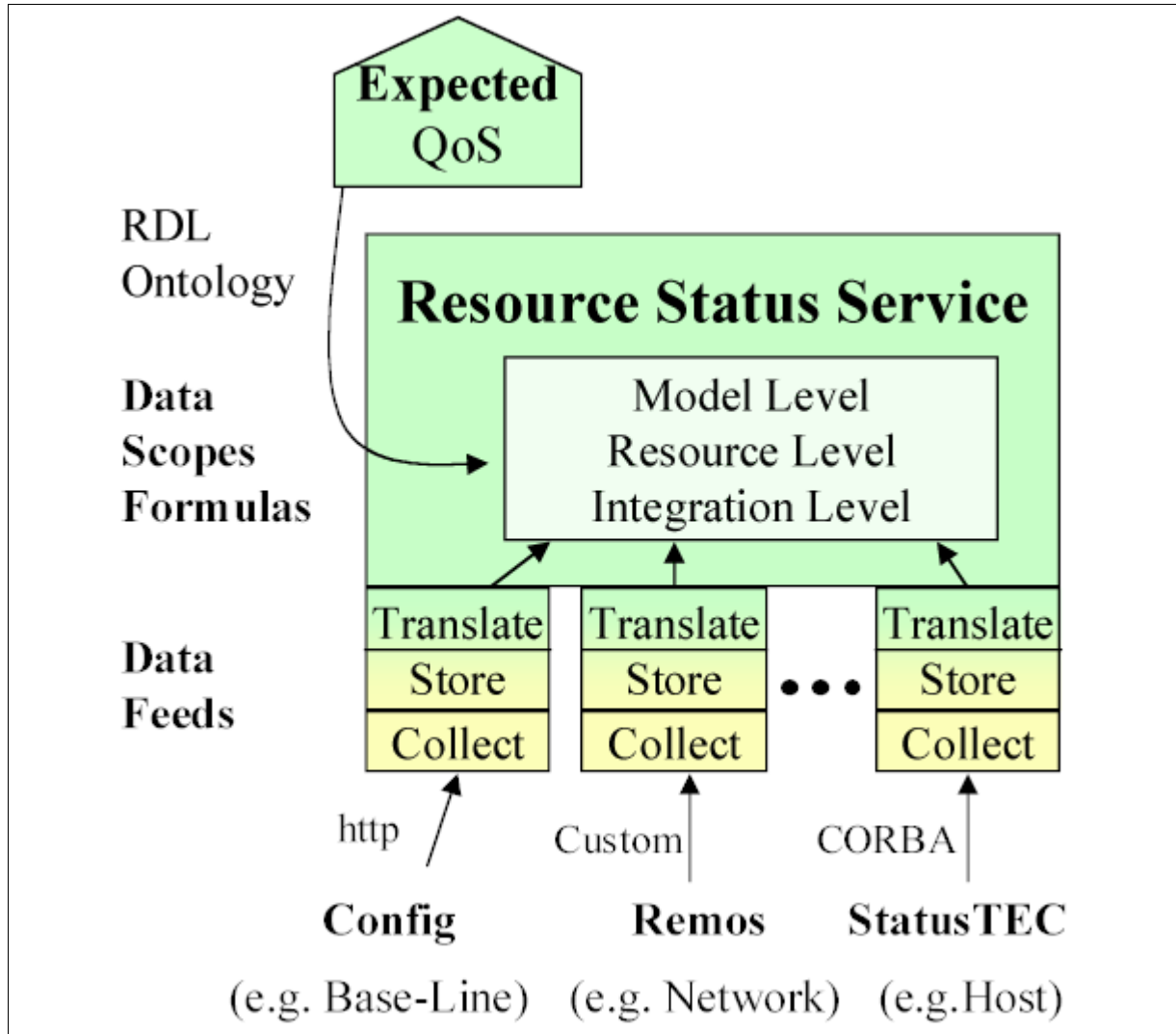
*QuO Uses Both In-band and Out-of-band Mechanisms to Observe Measured and Expected Quality of Service*

Managing the out-of-band status data is a tedious and difficult job. For example, many different sources of status information exist, each having their own syntax and semantics. The sources are distributed throughout the networks, so that collecting the information is as complex as the distributed application itself. To get useful predictions of expected performance the gaps in the data must be filled with the best guess.

QuO RSS is a prototype for a comprehensive system for performing this job. The goal of the QuO RSS is to make it easy for the adaptive code to access status information about the underlying resource. This will improve the quality of the adoption and reduce the cost to develop the adaptation. Also, the RSS can be reused across many applications and configurations.

## 9.2 RSS Architecture

The RSS is a unified way to access information about the expected behavior for the system and its resources. The RSS takes care of the whole local process for obtaining this information. Because resource status is used on demand, it is gathered in anticipation of its use and cached for quick access.



*Resource Status Service Architecture*

The RSS architecture breaks the job up into two basic components one that gathers raw status information and a layered knowledge-base which is updated from the raw information (see figure above). The RSS collects resource status information directly from the resources or their managers and may use several sources information. Because each source has different formats and semantics, the raw data is translated into a common internal format. The data from the different sources are integrated into a single view. Finally, QoS models can predict the expected behavior of the system and its underlying resources. The RSS has the following architectural components.

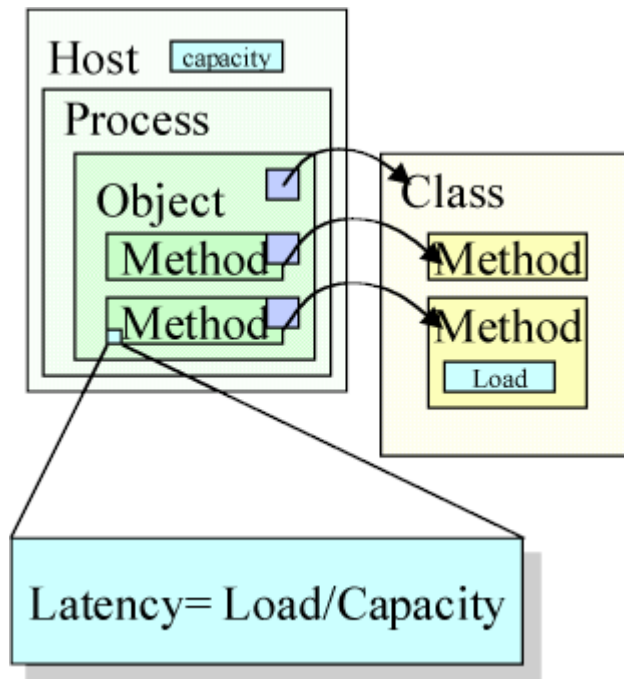
### **9.2.1 Data Feed**

Data Feeds manage gathering raw information from a single source. The Data feed communicates directly with the source and must use its protocol. This may involve linking in a source-specific library, such as using a standard protocol, such as CORBA, RMI, or HTTP. Because the source is remote, the Feed may cache the raw values for easy access or in anticipation of future requests. The Feed also translates the source's data format and semantics into a common ontology that is used throughout the RSS. Feeds support a subscription interface, which allows Data Formulas to get translated data from the Feed.

The QuO Version 3.1 supports several types of Data Feeds. The Property Data Feed can remotely load property files for the base configuration of the system. The Status TEC Feed subscribes to a QuO Status Typed Event Channel, which uses push technology to publish the status of hosts. The Sysstat Data Feed provides local host and process data. Together these feeds give enough raw information to effectively model the basic resource status at the granularity needed for application adaptation.

### **9.2.2 Resource Contexts**

Resource Contexts are the basic objects in the knowledge base. Resource Contexts form a containment hierarchy, which models the underlying resource base. Resource Contexts are created on demand, so only the parts of the resource base for which that application is interested is instantiated. Different types of Resource Contexts model different components. For example, a Host context may contain a Process context and a Process context may contain an Object context. Resource Contexts have parameters, which are, specified when the Resource Context is created and used to distinguish it from other contexts of the same type. A Resource Context can also have Data Formulas, which calculate a value dynamically. Resource Contexts can be linked by other relationships, such as proto-type and connect.



*Resource Contexts Form a Hierarchy of Containment and Other Relationships*

To access an attribute of a Resource Context, the mesh of relationships is traversed until a parameter or formula is found that matches the attribute. For example, accessing the “Load Average” attribute for an Object would follow the containment hierarchy from Object to Process to Host (see figure above).

### **9.2.3 Data Formula**

Data Formulas dynamically calculate the value for an attribute of a Resource Context. Formulas form the publish and subscribe network, where if the one of the raw values change, all the formulas that depends on it will recalculate their values. Data Formulas can also be queried for their values that may forward chain to calculate its value. Data Formulas can also cache their values for quick access.

### **9.2.4 Data Value**

The RSS and Status TEC do not maintain just a simple value for an attribute. A Data Value is really a bundle of information, which can include the source of the data, when it was collected, how it was collected, and the data units. The Data Value supports the notion of *credibility*, which is how much faith you have that the data is right. Credibility summarizes many factors, such as staleness, trust of the source, measurement technique, correlation with other data sources, etc. For example, a compile-time default has less credibility than a network management configuration file, which has less credibility than a direct measurement. Credibility is used to integrate data from many feeds, *i.e.*, taking the data value with the highest credibility.

### 9.3 QuO Resource Ontology

The RSS has a prototype for Resource Ontology. Much ontology for resources exists, such as DMTF's CIM, or SNMP MIBS. The QuO Resource Ontology is loosely based on these ontologies, but only implements a minimalist set of objects and attributes. Improving this ontology is left for future study.

The ontology is a simple tree with several roots and values at the leaves, *i.e.*, much like an SNMP MIB. The branch names are simple strings with an underscore (“\_”) as the separator. The base ontology is:

```
IP_Flow_<srcIpAddress>_<dstIpAddress>_
  Capacity_Max_Remots
  Capacity_Unused_Remots

Site_Flow_<srcIpAddress>/<mask>_<dstIpAddress>/<mask>_
  Capacity_Max_Config
  Capacity_Unused_Config

Host_<ipaddress>_
  CPU_loadavg_ProcessStats
  CPU_Jips_ProcessStats
  CPU_bogomips_ProcessStats
  CPU_count_ProcessStats
  CPU_MHz_ProcessStats
  CPU_cache_ProcessStats
  CPU_bogomips_ProcessStats
  CPU_count_ProcessStats

Network_TCP_sockets_inuse_ProcessStats
Network_UDP_sockets_inuse_ProcessStats

Memory_Physical_Total_ProcessStats
Memory_Physical_Free_ProcessStats
```

### 9.4 Setting up the RSS

The RSS depends on having data sources for status information and defaults. QuO V3.1 supports three types of feeds, each with their own server setup procedure. The easiest to step up is the static resource configuration. This involves setting up a web page with a property files. The next easiest is setting up the QuO Status TEC, which collects hosts status, but needs a Java-based process running on each host.

When the application runs, it must be given a QuO Kernel configuration file which specifies which feeds to start up. An example startup file can be found in \$QUO\_ROOT/examples/bette/kenel.conf.

---

### 9.4.1 *Setting up Resource Configuration Servers*

Setting up a resource configuration involves editing a text file and publishing it on a web server. We recommend breaking up the configuration into three separate files, each for a different type of data, as follows:

**Sites.conf** should list the capacity between subnets and individual host pairs. The form of the records should be:

```
Site_Flow_<srcIpAddress>/<mask>_<dstIpAddress>/<mask>_Capacity_Max_Config_value=<bps>
```

**Hosts.conf** should have the default capacity for each of the hosts in your environment. The form of the records should be:

```
Host_<ipaddress>_CPU_Jips_ProcessStats_value=<jips>
```

**BetteAppl.conf** is a description (calibration) of example bette application (for both CORBA and RMI). The file can be found in \$QUO\_ROOT/examples/bette/BetteAppl.conf

### 9.4.2 *Setting up StatusTEC*

The QuO Status TEC consists of two system-wide servers and probes on all your host. The QuO 3.1 release contains rpms for probes and servers. Documentation for setting up the Status TEC is in \$QUO\_ROOT/dirm/event

## 10 Integrating QuO Services

QoS Mechanism Designer builds QoS adaptive code into the underlying infrastructure, such as DiffServ prioritization, CPU schedulers, or TAO real-time ORB. The QuO architecture has many hooks for integrating this adaptive code. Once a SysCond wraps the service so that it is accessible, Qoskets can use the SysConds to implement reusable adaptive code. Hence, QuO is good glue for binding new QoS mechanisms to applications.

The job of creating a new QuO service is to wrap the infrastructure in a way that QuO SysConds can access it. The basic techniques are to wrap the server controls with a CORBA interface, which makes it easy to call from a QuO SysCond. Or a SysCond can be developed that calls the service directly. If the service is only publishing monitoring information, then it can be integrated into the RSS. We will discuss each of these techniques below and point to example code.

### 10.1 Creating CORBA Servers

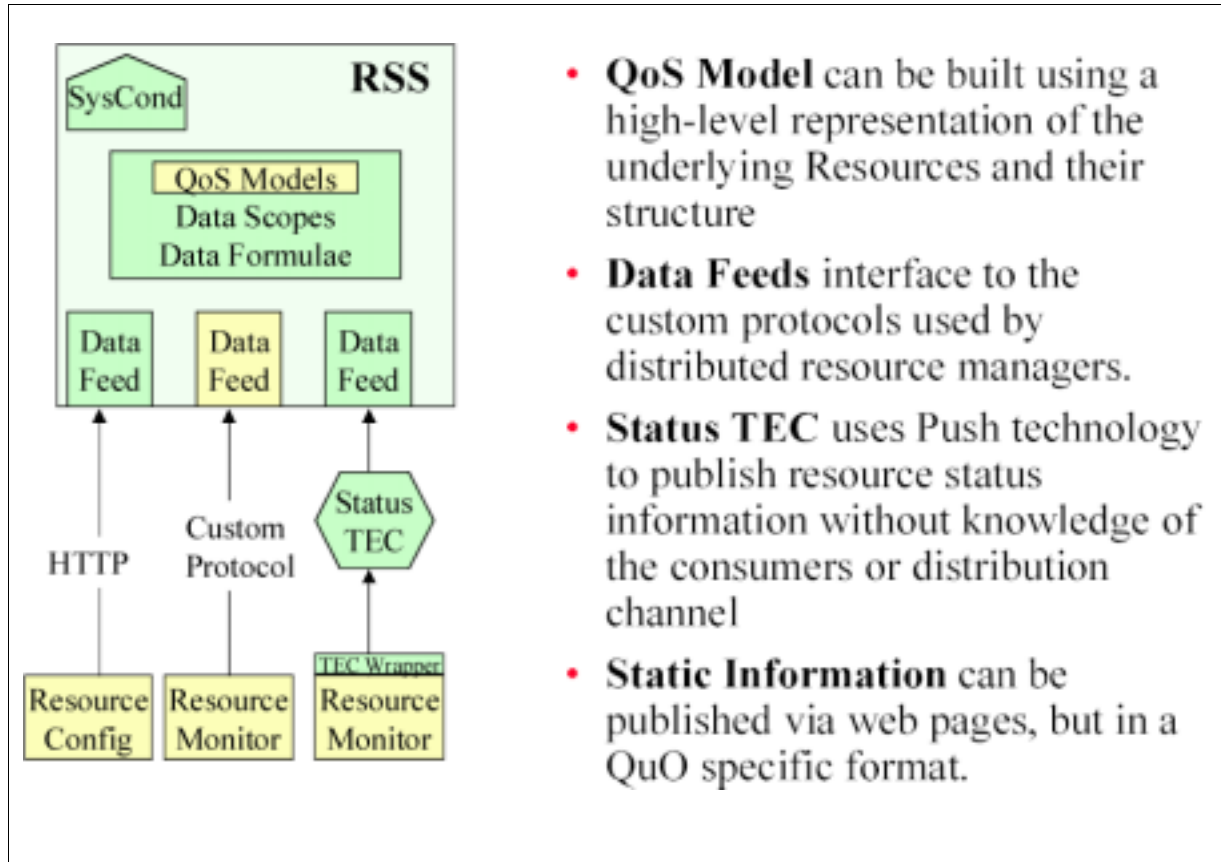
The easiest way to get access to remote services in a heterogeneous language and OS environment is to wrap the service with a CORBA object. CORBA objects can be polled by extending the MonitorSCImpl, which is part of the QuO Kernel.

### 10.2 Creating SysConds

If the service gives a library as its interface to the service, then a SysCond can be created to access that library

### 10.3 RSS Integration Points

The RSS introduces three new integration points to the QuO architecture (see figure below). Static configuration information can be published via web pages, but the data must be in a QuO specific format, i.e., a property file using the QuO RSS path names. Dynamic monitoring information can be published onto the Status TEC. A good example for how to publish data into the Status TEC is the netmap publisher in the \$QUO\_ROOT/dirm/event service. If your status service generates lots of data, of which only a small part is used by any one client, then a query/pull interface is necessary. For query-based data sources, a new DataFeed is the best integration point. Both \$QUO\_ROOT/dirm/event has example DataFeed code and feed testing examples. Additionally, \$QUO\_ROOT/erni has example DataFeed code and feed testing examples as well.



*The RSS Supports Three Integration Points for Publishing Resource Status*

*This page is blank intentionally.*